The often specialized information processing requirements of the federal government are of particular interest to the Federal Systems Division of IBM, which has been heavily involved in the development of complex, on-line, real-time systems for diverse government applications for over twenty-five years. In the paper by Olsen and Orrange, a representative sample of these programs is reviewed. These range from an early air defense system of the mid-1950s to a current modern naval light airborne multi-purpose system. The key issues characterizing real-time systems hardware and software development and their technological implications are discussed; significant aspects of real-time processing methodology, reliability and availability, and total systems responsibility are reviewed; and an evaluation of the lessons learned from this wealth of experience is presented.

A companion paper by James traces the evolution of real-time computer systems developed by IBM to support the U.S. manned spaceflight program since its inception. This paper provides an overview of approximately twenty years of development of command and control systems for NASA's Real-Time Computer Complexes (RTCCs). The emphasis here is on the development of RTCC systems, the tools and techniques used in the development process, the technological and architectural changes influencing this development, and the experience gained in the management of complex real-time programming systems.

The paper by Taylor is concerned with the IBM products referred to as small general-purpose systems: the System/3, System 32, System/34, and System/38 series of machines. These systems were originally intended primarily for first-time users, and there have been extensive enhancements to these products in response to expanding customer requirements and to very challenging objectives regarding cost/performance, ease of use, and applications development support.

A companion paper on small systems, by Harrison *et al.*, traces the highlights of the evolution of small real-time IBM computers designed initially for sensor-based and industrial-control type applications. These are the 1720, 1710, 1800, System/7, and the Series/1 line of machines. Over time, their applications scope has broadened to include diverse operational environments such as discrete manufacturing control, laboratory automation, traffic control, and energy management systems.

In the concluding paper of this chapter, Hsiao *et al.* review the significant RAS improvements incorporated in IBM computer systems over the past quarter century. These RAS enhancements have been motivated by the considerable increases in function, number of components, and complexity, and by the ever-growing dependence on computers in the day-to-day operations of customer enterprises. The authors trace the evolution of computer RAS technology and methodology in IBM systems, and show how RAS capabilities have been enhanced through continual reduction in component failure rates and through improvements in system error detection, error recovery, and serviceability characteristics.

Editor

C. J. Bashe
W. Buchholz
G. V. Hawkins
J. J. Ingram
N. Rochester

The Architecture of IBM's Early Computers

Most of the early computers made by IBM for commercial production are briefly described with an emphasis on architecture and performance. The description covers a period of fifteen years, starting with the design of an experimental machine in 1949 and extending to, but not including, the announcement in 1964 of System/360.

Introduction

This is a technical account of IBM's principal computers during the fifteen-year period beginning in late 1949 and leading up to the announcement in 1964 of System/360. They were all electronic stored-program computers. Each had a random-access memory that was reasonably large by the standards of the time, could do arithmetic and logic, could call subroutines, could—and did—translate from various symbolic languages to its own machine language, and each had a substantial set of input/output (I/O) equipment. This account emphasizes architecture and performance, that is, the programmer's view at the level of machine language.

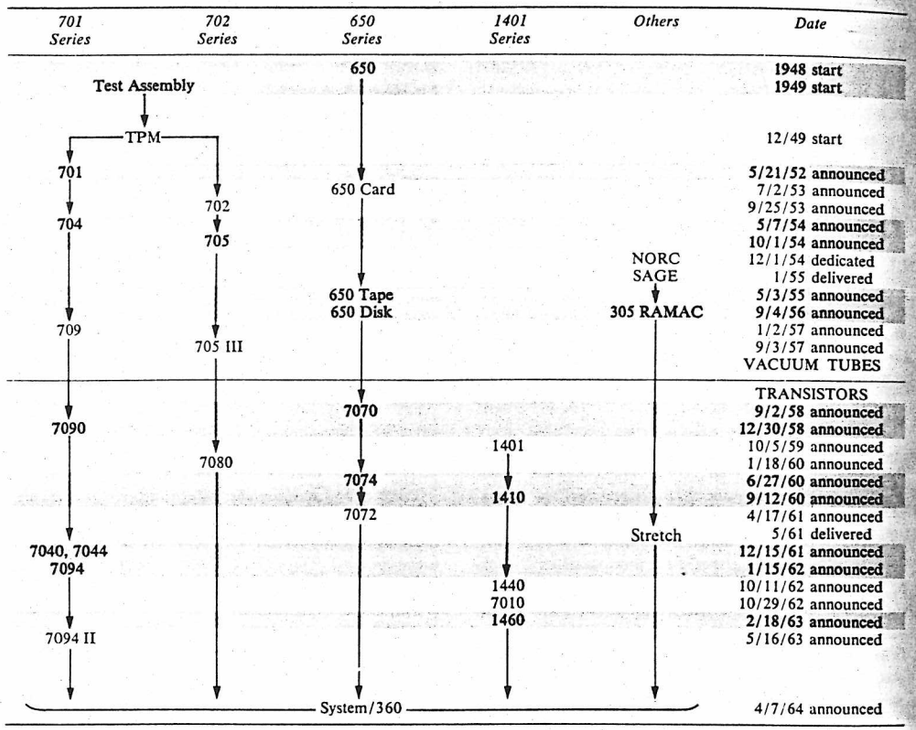
The paper begins with a section on machines which were precursors of the computer era. This is followed by descriptions of several series of related computers, each named after its first member: 701, 702, 650, and 1401. Next is an essay on the IBM Stretch system, and the final section concerns I/O techniques for all these machines.

Not all of IBM's stored-program computers of this period are included; because of space limitations, several interesting and important machines are omitted. The selection is based on a subjective judgment of which machines had technically the most impact during those fifteen years and beyond.

Table 1 illustrates the chronology of the machines discussed in this paper. The members of a family or series of machines appear vertically in a column; the later entries are more or less upward compatible with earlier ones. (There is no family relationship, however, among the machines in the column labeled "Others.") All machines up to 1957 used vacuum-tube electronics; from 1958 on, all of IBM's computers were built with transistors. Every machine listed is discussed or referred to in the text.

Except for a discussion of IBM's precursor machines, no attempt is made here to acknowledge adequately the many pioneering projects and individuals in IBM and elsewhere which made these developments possible. The 701 clearly owed much to the machine at the Institute for Advanced Studies, to prior work at the University of Pennsylvania, and to Whirlwind at MIT; the 702 was largely stimulated by the Eckert-Mauchly UNIVAC; and the 650 had early roots in work at Engineering Research Associates. Cathode-ray tube (CRT) memory, as used in the 701 and 702, and index registers are well known to have originated at the University of Manchester. The names of J. P. Eckert and J. W. Mauchly, of J. von Neumann, who was a consultant on the 701, and of F. C. Williams readily come to mind. To give sufficient credit beyond the obvious, however, would go outside the scope of this paper.

Table 1 Chronology of principal IBM computers before System/360.



Precursors

The decade of the 1950s was a significant one in the history of IBM computers. All of the electronic stored-program computers designed for regular production were announced and first delivered after 1950. All earlier devices that were more than electromechanical accounting machines were either one-of-a-kind machines or combinations of electromechanical and electronic calculating units, which did not incorporate an electronic form of stored program [1].

There were, to be sure, earlier examples of the greater logical power obtained from large numbers of calculating elements operating under elaborate sequence controls and of the increased speed resulting from substitution of electronic circuits for electromechanical relays and counters. The IBM Automatic Sequence Controlled Calculator, which was presented by IBM to Harvard University in 1944 and became known as the Harvard Mark I, was based on electromechanical devices for both calculation and sequence control. The 604 electronic calculating punch, which IBM started shipping in 1948, used electronic counters for calculation and was controlled by a plugboard program of up to 60 three-address instructions.

A major step was the dedication, also in 1948, of the Selective Sequence Electronic Calculator (SSEC) at IBM's headquarters in New York City. It had a hierarchical memory system consisting of 20 000 words of paper-tape storage, 150 words of relay memory, and eight words of electronic memory, each word consisting of 19 digits and a sign. Its electronic arithmetic unit could add, subtract, divide, and multiply, the latter in 20 ms. The paper-tape storage consisted of 66 readers, each holding a loop of tape as wide and thick as an IBM card. These tapes could hold tables, data to be operated on, or sequences of 78-bit instructions. The SSEC was the first operating computer capable of treating its own stored instructions exactly like data, modifying them, and acting on the result.

In 1949 IBM introduced the Card Programmed Calculator (CPC), the concept for which originated at Northrop Aircraft. The CPC had a 604 as its arithmetic unit, a card reader to read the program from punched cards, a 480-digit electromechanical memory, a card punch, and a tabulating machine to print the output. It required a sorter, a collator, and a cart to move decks of cards around. This machine provided the means to do computing on a hitherto impractical scale, enabled users to develop computing methods and expertise, and increased the awareness of and interest in the potential of fully electronic, stored-program computers, which were just coming into existence.

The final precursor, the Test Assembly, was such a computer. This experimental unit was built in the IBM Poughkeepsie laboratory starting in 1949. It used a 604 as its electronic arithmetic unit, a second machine frame with 604-type circuits as logical elements to exercise control, a 250-word CRT memory that was a prototype of the 701 and 702 memories, a magnetic drum memory to store both data and instructions, and a card reader-punch for I/O. By solving a differential equation in 1950, the Test Assembly demonstrated that these components could be used to build a full-fledged, stored-program computer.

The 701 series

The 701 was the first large-scale electronic computer that IBM put into production. To make the machine quicker and easier to design, implement, and maintain, the architecture was kept simple and spartan. Many desirable features, such as floating-point arithmetic, were left to be done by programming. One quarter of the central processing unit (CPU) remained empty to give flexibility for correcting errors or adding function. The successor of the 701, the 704, provided the added function and filled up the space. Then there followed a succession of larger and faster, upward-compatible machines with new features as they were invented, culminating in the 7094 II and its smaller, simplified offshoots, the 7040 and 7044.

- *The 701*

The 701 was a parallel, binary, single-address, stored-program computer with a CRT memory of 2048 (later 4096) words of 36 bits [2–5]. It had 33 18-bit instructions with which to perform arithmetic and logic operations, sense and control its environment, and direct, word by word, the flow of information to and from

- two pairs of 1250-word-per-second magnetic-tape drives, adapted from designs developed for the TPM (702) project,
- four 2048-word magnetic drums, adapted from early designs for the 650 computer,
- a 150-line-per-minute printer,
- a 150-card-per-minute card reader, and
- a 100-card-per-minute card punch.

Instructions could address 18-bit half words or 36-bit full words. The instruction set was simple but carefully polished for efficiency and consistency. A 36-bit binary addition took five 12-μs cycles, while multiply and divide instructions each took 38 cycles.

The 701 was made binary, instead of decimal, because fast binary arithmetic was simpler and therefore less costly, yet it did not mean giving up the advantages of decimal and alphabetic input and output. Conversion between external decimal and internal binary representations and between punched-card and magnetic-tape codes could readily be handled by programming in a binary machine and did not require special translation circuits. The choice of binary arithmetic with its superior characteristics right at the start also set an important precedent: It might have been very difficult to introduce a binary machine later on if all of IBM's early computers had been decimal.

- *The 704*

In the 704, magnetic-core memory with from 4096 to 32 768 words replaced the CRT memory of the 701, thereby eliminating the most difficult maintenance problem and providing users with a more efficient way to run large programs. In the early 1950s, there was a widely held belief among theorists that 1000 words or so of memory should suffice. However, practical programmers seemed to need more than that. Also, the cost of providing larger memories kept going down as engineering and production methods advanced.

To accommodate the larger address space and other requirements, the instruction length was expanded to 36 bits. The instructions were similar in function to the 701 instructions but were incompatible, and their number was increased to 91. Floating-point arithmetic was provided. Three index registers were added for automatic address modification and for programming iteration loops. The combination of built-in floating point and indexing greatly improved performance.

An instruction for entering a subroutine put its location in an index register before branching. The subroutine could use indexed instructions to access parameters in

the main routine and an indexed branch to return. This hardware implementation of a programming technique invented at the University of Cambridge made subroutines much more efficient and has become widely used. Many other instructions, too detailed to name here, contributed to the overall efficiency.

A CRT display was available, primarily as a graphic output device.

- *The 709 and 7090*

The 709 was basically a 704 with several additional architectural functions [6]. The most important of these were its data-synchronizer units (now called channels), which relieved the CPU of the need to control the transmission of individual words between memory and I/O units.

The 7090 was a transistorized version of the 709 with some added facilities. Like the change from CRT to magnetic-core memory, the change from vacuum tubes to transistors was one of technology and not of architecture. However, the effect of these two changes was so profound that it must be mentioned in the context of architecture. The 7090 was not just less costly to maintain and more reliable than the 701: it was so much better that it was a qualitatively different kind of machine.

The set of more than 180 instructions of the 709 and 7090 provided various other improvements. Among these were three convert instructions, which speeded up the translation from one number base to another and other kinds of access to tables of information.

- *The 7094*

In the 7094 the multiply and divide circuits were reworked to increase the speed of fixed- and floating-point operations and to provide a full set of double-precision floating-point arithmetic instructions. New instructions were added, bringing the total to 274.

The performance of the 701 series culminated in the 7094 II, which had a shorter cycle time, fewer cycles to multiply and divide, and memory interleaving. Fixed-point multiplication was 108 times faster than in the 701, and the speed on a problem using double-precision floating point had increased by a much larger factor.

- *The 7040 and 7044*

These machines provided a less costly alternative to the 7090 and 7094 for users whose requirements could be met by a simpler machine. They had a basic subset of 49 instructions as well as six sets of optional instructions that could bring the total up to 120 as compared to 180 on the 7090 and 274 on the 7094. The machines differed only in speed, with an 8-μs cycle on the 7040 and a 2.5-μs cycle on the 7044. A few 7040/7044 instructions were not part of the 7094 instruction set; thus, writing compatible programs required avoiding these improvements.

A novel combination, the Direct-Coupled System (DCS), was an early asymmetrical multiprocessing system that was very successful. It used a 7040 or 7044 as a front-end unit to handle I/O and to schedule jobs, and a 7090 or 7094 to do the computing. Combinations similar to DCS are still in common use with System/370 computers.

- *Special system for CTSS*

A particularly significant modified 7090, and later 7094, made it possible to create the first major time-sharing system: the Compatible Time-Sharing System (CTSS) at M.I.T. A second 32 768-word memory bank was provided. User programs occupied one bank while the operating system resided in the other. The user memory was divided into 128 memory-protected, 256-word blocks, and a user could not refer to a block outside of his allocation. The contents of a 7-bit relocation register were added to the most significant bits of each address, so that each user programmed as if his allocation started at address zero. There was an interrupt clock that enabled the system to control and record the time allocated to each user.

The 702 series

The 702 project started in late 1949, when design was begun in the IBM Poughkeepsie Laboratory on an engineering model, then called the Tape Processing Machine (TPM). Work was slowed considerably in 1951, when much of the TPM engineering staff was assigned to the higher-priority "Defense Calculator" (701) project because of the Korean War. The TPM model became operational in 1952; it was revised and announced as the 702 in 1953. Thus, the 702 project started earlier than the 701 but finished later.

- *The TPM and 702*

As its name implied, the Tape Processing Machine was intended to provide a data processing system with the speed and functional advantages of a new storage medium, magnetic tape, replacing punched cards. Its decimal arithmetic and logical unit, CRT memory, and magnetic-tape units are described in detail in a patent filed in 1954 and issued in 1966 [7].

The TPM was substantially redesigned to achieve greater performance and reliability, its instruction set was revised while retaining much of the basic architecture, and it became the 702 [8, 9]. Serial-by-bit, serial-by-character operation was changed to parallel-by-bit, serial-by-character operation, which turned out to provide higher performance at lower cost. The single-address instructions operated on one operand in CRT memory and another operand in a second CRT storage device called an accumulator. Instruction execution times of the 702 varied greatly depending on the number of characters involved. Numbers of five decimal digits could be added in 0.25 ms, whereas multiplying ten by ten digits took 3.36 ms.

Several significant innovations especially useful for business data processing were introduced by the TPM and 702. These are now briefly discussed.

Variable field and record length In contrast to the 36-bit word orientation of the machines that formed the 701 series, the architecture of the 702 series was oriented towards alphanumeric characters. A character was encoded as six bits and an even-parity check bit that was used to detect data errors going to and from memory and on tape. An instruction could directly address any character in memory and process a field of arbitrary length; both the memory operand and the operand in an accumulator could be up to 511 characters long. Likewise, records on magnetic tape could be of arbitrary length. One disadvantage was that fields and records were delimited by codes within the data, which imposed restrictions on the character set. This data-marking technique was subsequently replaced, in Stretch and in System/360, by having the program specify lengths explicitly.

Collating sequence Sorting and merging of files of alphanumeric data require an appropriate "collating sequence," which governs the low-to-high sequencing of alphabetic, numeric, and other characters. A compare instruction was designed to provide a particular collating sequence which was already well established for punched-card collators, so that existing file sequences could be maintained when converting from punched cards to magnetic tape.

Editing instructions Ease of data editing was a major criterion for selecting the 702 instruction set in general and variable field length in particular. One special instruction inserted commas and periods and suppressed leading zeros in numeric fields; another instruction facilitated the insertion of floating dollar signs and check-protection asterisks. Although the card reading, punching, and printing units were derived from standard punched-card equipment on which editing was controlled by plugboard wiring, their plugboard facilities were omitted on the 702 devices. All editing was to be done by programming so as to minimize operator intervention.

Competent I/O Most early non-IBM computers had only primitive I/O equipment, such as paper-tape readers and punches, or magnetic-tape devices connected to keyboards and typewriters. It was considered essential for business applications to provide more power and flexibility: high-speed magnetic tape for file storage, card readers for efficient data entry, card punches for producing machine-readable output documents, and high-quality line printers at least equal to the best in use in punched-card installations. The variety of I/O needed for different applications required flexibility in attaching the I/O devices; this aspect is discussed in a later section.

It soon became evident that CPU speed was more important in a data-processing environment than had been thought at first. The "housekeeping" required to keep just the I/O running at full speed consumed a great deal of CPU time. Improved handling of large tape files was also a concern, because much machine time was being spent on sorting, merging, and searching of files. The need for more speed and the desire to replace the CRT memory and accumulator storage with faster and more reliable magnetic cores prompted an early development of a 702 successor. At the same time, considerable effort was spent on developing an auxiliary tape sorting and collating machine, tentatively called the 703, which was intended to relieve the main computer of much of its tape-handling burden.

- *The 705 models*

The first 702 successor was the 705, announced in 1954, which provided the following significant improvements:

- The memory size was increased from 10 000 characters to 20 000 (705 I) and later to 40 000 (705 II). The 702 addressing scheme had to be modified to accommodate the larger memory.
- All the variable-field-length instructions with addressing of individual characters were retained, but instruction fetching and data moving within memory by means of some new instructions were done five characters at a time.
- Tape reading could be overlapped with tape writing.

The redesign and the nonvolatile core memory provided substantial performance improvements. Addition of five digits took 119 μs, and ten-by-ten multiplication used 2.41 ms. The 705 also proved to be sufficiently better than the 702 at handling tape that the plan to announce the specialized 703 tape sorter-collator was dropped.

Although the 705 instruction set was largely the same as that of the 702, it was different enough that 702 programs could not be run on the 705. With the successors to the first 705 model, however, compatibility became an important consideration because of the inconvenience to

users of having to convert existing programs. Even minor differences, such as new functions which only made use of previously unassigned bits of an instruction, were found to cause problems, and a "compatibility mode" was needed to be able to run programs that had used these bits for their own purposes. The conversion problem was more significant than it would be today, because high-level languages were still in their infancy, and most programming was done at or close to the machine-language level.

The 705 III, announced in 1957, provided up to 80 000 characters of core memory, indirect addressing, and one or more "data-synchronizer" units to permit overlap of tape reading and writing with computing, using main memory as the buffer. It added five digits in 86.5 μs and did ten-by-ten multiplication in 1.62 ms. Several instructions were added to do bit handling, to simplify address computations, and to blank a memory area. The 705 III was capable of running 705 I and 705 II programs with only minor exceptions.

- *The 7080*

The 7080 was a transistorized version of the 705 III. It provided up to 160 000 characters of memory, a set of I/O channels to replace multiple data-synchronizer units, a single-level interrupt facility to handle I/O, and compatibility modes to run 705 I–II or 705 III programs. It could perform a five-digit addition in 11 μs and a ten-by-ten multiplication in 265 μs.

The 650 series

- *The 650*

In 1948, when the engineers from the IBM Endicott laboratory who designed the SSEC had completed that project, they turned their attention to the design of a new and much smaller computer that would be suitable for volume production. From the outset, the emphasis was on reliability and moderate cost, which led to the choice of a magnetic drum as the central storage medium and to a high degree of checking throughout. The result was the 650, which was announced in July 1953 [10, 11].

The 650 was a serial, decimal, stored-program computer. A word, which had a fixed length of ten decimal digits and a sign, could contain one instruction or one decimal number. Decimal digits were represented internally in a seven-bit (biquinary) self-checking code, and on the drum in a five-bit (2-out-of-5) self-checking code.

The magnetic drum, which had a capacity of 2000 words, rotated at the unusually high speed of 12 500 rpm. The drum also served as an I/O buffer for the card reader and punch, the only I/O units provided initially. The high speed of the drum kept the maximum time of access to instructions or data to 4.8 ms. To allow still further reduction of the effective access time, a two-address instruction format was chosen, one address for the operand and the other for the next instruction to be executed. With the aid of the Symbolic Optimal Assembly Program (SOAP) developed for the 650, a program could be optimized so that the data or the next instruction would be close to the read-write heads at just the time they were needed, thus avoiding a long wait of up to one whole revolution for each. Such optimum programming could bring the time for an add instruction from an average of 5.2 ms down towards the minimum of 0.8 ms.

The basic 650 with its card equipment was very successful. Later it was expanded with alphabetic capability, using two digits to represent one alphanumeric character, and with on-line printing and magnetic tape. Also, a 600-digit core storage was added as a high-speed I/O buffer.

The most important addition was a magnetic disk unit, the high-capacity storage device that had been developed at the IBM San Jose laboratory as an integral part of a smaller machine, the 305 RAMAC [12]. The 650, whose version of the disk unit had a storage capacity of six million digits, thus led the way in attaching disks to general-purpose computer systems. The 650 also acquired terminals, primarily for inquiry and response applications. This combination of magnetic disks and terminals was important as it constituted the beginning of a major shift from tape to disk for secondary storage and from batch to transaction-oriented processing.

With the addition of these I/O capabilities, the 650 was no longer solely a card-processing system but had become an intermediate data-processing system with considerable flexibility.

- *The 7070*

The successor to the 650 was the 7070, which was a transistorized machine with up to 9990 ten-digit words of core memory [13, 14]. Because of the random-access core memory, the two-address instruction format of the 650 was no longer needed and was replaced by a single-address format. Decimal digits were represented in a five-bit (2-out-of-5) code throughout. To make these basic changes possible, the 7070 was not constrained to be compatible with the 650.

By omitting the second address in an instruction, space was freed up for two additional functions. Two digits were used in indexing to select one of 99 index words located at the low end of memory. Two more digits constituted the field definition, which allowed a portion of a ten-digit word to be processed by a single instruction.

The 7070 transferred words in parallel between memory and the CPU, but it processed them serially, digit by digit. Addition of two ten-digit numbers required 72 μs, multiplication 924 μs. Floating-point decimal arithmetic was also available.

To permit overlap of CPU processing with multiple input and output operations, the 7070 had an I/O interruption capability called priority processing. It allowed a main program to be interrupted when an I/O device required service, which was provided by switching to a "priority routine" that was not interruptible. Priority processing was used to perform peripheral operations, such as card-to-tape and tape-to-printer conversion, on the main computer instead of an auxiliary system. The term SPOOLing (Simultaneous Peripheral Operation On Line) was coined for this function, which represented an early form of multiprogramming.

A block-transmission facility provided for movement of blocks of data, both within memory and between memory and I/O, in a single operation. The blocks, which did not have to be in contiguous memory locations, were described by a chain of record-definition words.

- *The 7074 and 7072*

The 7074 was a faster version of the 7070, with storage increased up to 30 000 words [15]. It had the same instruction set except that, when using the additional storage, the 7074 had to be switched to a different addressing mode. Parallel arithmetic allowed the 7074 to add ten-digit numbers in 10 μs and multiply them in 56 μs, considerably faster than the 7070. The 7072 was a lower-cost, tape-only version of the 7074 oriented towards numerical applications.

The 1401 series

The upward growth of the 650 series and the rapid progress in semiconductor technology during the mid-1950s left a void at the low end, which was filled by the 1401 series. The 1401 was initially intended as a stand-alone stored-program computer to provide faster card reading, card punching, and printing equipment than did earlier electromechanical punched-card machines. Its chain printer, which combined a high speed of 600 lines per minute with high print quality, was a technological breakthrough. This printer, with some modifications, is still a mainstay of the IBM product line. Since the high-speed reading, punching, and printing devices of the 1401 were also valuable to users of the larger 7000-series computers, it was decided to provide a magnetic-tape version as an auxiliary system for off-line card-to-tape, tape-to-card, and tape-to-printer operations. Both versions of the 1401, which were announced at the same time, became eminently successful.

- *The 1401*

The architecture of the 1401 was character-oriented and resembled that of the 702 in several respects, but there were important differences. The 1401 used two-address, memory-to-memory instructions; data flow was serial-by-character, with memory cycles alternating between the two operands. Both data and instructions were variable in length. Characters were represented by eight bits: six for data, one for odd-parity checking, and one for a "word mark" used to define the high-order position of each word. Data movement and manipulation were checked on all operations: thus numeric data were converted to a self-checking biquinary code before any arithmetic operation.

The 1401 had 34 different operation codes. An additional character could modify many of these operations, so that the total number was effectively much larger than 34. Operation codes were one character in length and addresses three; a typical two-address instruction was seven characters long. Not all instructions had two addresses: Some had no addresses and others one, so there were also one- and four-character instructions. Similarly, instructions with a modifier character could be two, five, or eight characters in length. "Chaining" of instructions was an interesting capability: Where consecutive instructions were used to add or move words located sequentially in storage, the first instruction with the usual two addresses could be followed by instructions consisting of only the operation codes.

Editing of numeric fields for printing was accomplished in a manner similar to the editing facility of the 702, but additional function and flexibility allowed each field to be edited completely with only one instruction.

The word mark, probably the most distinctive architectural feature of the 1401, in conjunction with the memory-to-memory logic, minimized the electronic circuitry requirements. Only two single-character data registers and three memory-address registers were used in the basic machine. Variable word length for both data and instructions reduced storage requirements very significantly.

Core memory was initially offered in capacities of 1400 (shown by studies to be adequate for card-accounting jobs) to 4000 characters. For more complex applications, the memory capacity was later increased to 16 000 characters. Addressing of 16 000 characters with a three-char-

acter address was accomplished by binary encoding of previously unused bits of the characters. Two other available address bits were used for indexing; that is, they could select one of three storage locations set aside as index registers.

The memory cycle time was 11.5 μs. Addition of five-digit fields took 230 μs, and multiplying two ten-digit fields required 6.9 ms.

The 1401 tape system was widely used, both as an auxiliary system for the larger computers and as an independent tape processor. Several special features were added to handle tapes for these different computers, thus bringing into focus the inconsistencies among them. Operation with even- and odd-parity tape was required. A column-binary feature allowed binary cards to be read or punched, and converted to or from tapes, for use on the 704/9 and 7090/4 machines. Similarly, special provisions were made to handle 7070 tapes. On the 1401 itself, the unique eighth word-mark bit could not be written directly on the seven-track tape that was common to all these machines, so this bit was translated to a word-separator character, which created another special tape format.

As applications for the 1401 expanded, numerous devices and optional features were added. Attachment of magnetic disk provided the third major 1401 version, the disk system, which was as well received as the card and tape systems. Processing could be overlapped with I/O, which gave significant performance improvement. Paper tapes, magnetic and optical character readers, and numerous other I/O media were attached via a serial input/output adapter, which also allowed channel-to-channel communication with other computers.

- *The 1440 and 1460*

The 1440 was a smaller, and the 1460 a larger, successor machine with the same architecture as the 1401 and with the same memory capacity of up to 16 000 characters. The 1440 and 1460 used newer circuits and had much of their electronics in common. The 1440 had an 11.1-μs memory cycle, giving it approximately the same internal speed as the 1401, but simpler and slower I/O. The 1460 with a 6-μs memory cycle had almost twice the processing speed, and up to three 1100-line-per-minute printers could be attached.

- *The 1410 and 7010*

The 1410 was a considerably larger and more powerful machine than the 1401. The memory cycle was reduced to 4.5 μs (4.0 μs optionally). The architecture was based on that of the 1401, but addresses were expanded to five characters to allow the memory capacity to be increased. Up to 80 000 characters of memory were provided. Zone bits were used to select one of 15 index registers. Two address registers were added to speed up arithmetic operations. The instruction set was enhanced considerably by expanding some instructions and adding other functions, such as table look-up. An optional I/O interruption feature, similar to that of the 7070, was available. The 1410 could run 1401 programs by switching to a 1401 compatibility mode.

At the top of the 1401 series was the 7010; it was fully compatible with the 1410. It had a memory of up to 100 000 characters, a 2.4-μs memory cycle, and two-character access to storage.

Stretch

The Stretch system, formally known as the 7030, played a special role in the evolution of IBM's computers [16]. It is best known as a "super-computer" developed for the Los Alamos Laboratory of the U.S. Atomic Energy Commission. A "Harvest" extension (the 7951) was designed for the National Security Agency [17]. Unlike IBM's earlier super-computer, the one-of-a-kind NORC (Naval Ordnance Research Computer) [18], the Stretch system went into production, though in limited numbers. More significant, however, was that much of the transistor, magnetic-core, circuit, and packaging technology of the Stretch project was carried forward to other 7000-series and 1401-series computers, and many of the architecture innovations were later applied to System/360.

In late 1954, a project was started at the IBM Poughkeepsie laboratory to develop a new, all-transistor computer, which was to benefit from the experience gained with the 701, 702, and 650 series that were marketed to all customers and with the SAGE real-time control computer [19, 20] that was marketed to the military. Subsequently it was decided to set the very ambitious goal of one hundred times the overall performance of the 704. The need for such performance was clear, as was the fact that the technology available at the time was not adequate. New transistors, circuits, magnetic cores, and design and manufacturing techniques would require the utmost technical effort, hence the name *Project Stretch*.

The technology which was developed especially for Stretch included drift-transistor circuits with a 20-ns delay time, multiple 128K-byte memory units (where K = 1024) with a 2.1-μs cycle time, and high-speed magnetic disks. This was combined with new organizational techniques, such as instruction look-ahead and memory interleaving. Error correction was used in memory and on the disks, and errors were checked extensively throughout the system.

Project Stretch introduced a number of new or substantially enhanced functions into computer architecture, and it brought new terms, such as "byte," "I/O interface," and the word "architecture" itself into the language of computers.

Addressing The architecture took advantage of binary addressing to provide storage subdivisions (words, bytes, and bits) of lengths that are powers of two. This choice greatly facilitated the design of variable field length, indexing, address arithmetic, and instruction formats. Addresses were up to 24 bits in length and permitted direct addressing of 262 144 or 2^{18} words of 64 bits (2048K bytes) in storage, and of any bit within those words.

Storage protection Protection against incorrect storage accesses by the program was provided by a pair of address-boundary registers.

Binary and decimal arithmetic Both binary and decimal arithmetic were included: binary arithmetic for speed and for address computation, and decimal for efficient data handling.

Variable field length Fixed-point arithmetic, comparison, and logical operations were provided for fields from 1 to 64 bits in length. They facilitated operations on decimal numbers, character strings, logical bit strings, single-bit indicators, and parts of instruction formats of various lengths, including binary addresses.

Variable byte size To operate on subdivisions (bytes) within a field, such as 4-bit decimal digits and 6-bit or 8-bit alphanumeric characters, variable-field-length instructions could specify a byte size of from 1 to 8 bits. For other than variable-length fields, the byte size was fixed at 8, a power of 2. The 8-bit byte also permitted a larger character set that included upper and lower case.

Floating-point arithmetic Much care was taken in the specification of the binary floating-point arithmetic to provide a complete, consistent, and easily programmed instruction set. Particular attention was paid to the control of precision loss and round-off errors, and to instructions which simplified the programming of multiple-precision arithmetic.

Indexing Each of the 16 index registers contained an index word of 64 bits, which provided not only the index value but also a loop count and a chain field. The combination could be used, among other things, to specify a chain of memory areas. I/O operations employed the same format for their control words. Consequently the same index or control word could be used to control reading of input data, then to index through the data during processing, and finally to control writing of the modified data to an output device.

Interrupts A single interrupt scheme provided a systematic method for responding to asynchronous I/O requests, time signals, and machine errors, as well as to synchronous program exceptions. The interruption conditions and other conditions which could not cause interruptions were all collected in one 64-bit indicator register. These bits could either be interrogated by a conditional branch instruction or, under the control of mask bits in a second register, cause execution of an instruction in an interrupt table. That instruction might be a branch to an interrupt handler or simply a one-instruction fix-up.

Clocks An interval timer provided an interrupt condition when a time interval had elapsed. A separate, continuously running clock permitted the programming of time-of-day indications.

Input/Output Up to 32 I/O devices could be operated simultaneously through the "exchange," a unit which corresponded to a 32-way byte-multiplexer channel in later terminology. Hardware to control data transfer was shared among many I/O devices. The exchange was completely device-independent, as were the controlling I/O instructions.

Consoles The operator console was connected via a channel and treated as an I/O device. Switches and lights were just so many bits of input and output, the meanings of which were determined entirely by the program. More than one console could be attached.

Multiprogramming Several of these architectural facilities—storage protection, interrupts, clocks, and program-interpreted multiple consoles—were intended specifically to make it possible and practical to use a large computer not only on single large problems, but also on multiple smaller programs simultaneously.

From the beginning the intent was to develop a very general architecture that would be well-suited to many different types of applications. It had been planned to make the high-speed, fixed-length arithmetic unit separable from the variable-length processing unit, so that the latter could subsequently be marketed as a lower-performance commercial product using the same instruction set. Indeed, some aspects of the architecture, such as variable field length, were primarily aimed at manipulation of alphanumeric data, since numeric computing had less need for them. Thus, it was thought that a single line of compatible machines, tentatively called the 7000 Series, would replace IBM's various earlier incompatible computers. This part of the architectural plan was not realized because of subsequent engineering and marketing decisions. Such a single line of architecturally compatible machines did not make its appearance until System/360 arrived.

For typical 704 programs, Stretch fell short of its performance target of one hundred times the 704, perhaps by a factor of two. In applications requiring the larger storage capacity and word length of Stretch, the performance factor probably exceeded one hundred, but comparisons are difficult because such problems were not often tackled on the 704.

It was decided to limit production to the Los Alamos machine (delivered in 1961), the Harvest system (delivered in 1962), and seven more systems that were on order at the time. Development of additional compatible versions was discontinued. Instead, the 7000 Series came to include various machines other than Stretch (7030); these used its architectural ideas and high-speed technology but not the plan for a line of machines with compatible architecture.

Early input/output architecture

The primary challenges of I/O architecture in IBM's early computers were quite similar to those which have faced system designers and architects up to the present day. These are

- To provide for efficient data flow between a very high-speed, precisely clocked processor/memory complex and a number of larger-capacity but (usually) slower and (usually) less precisely clocked I/O devices.
- To provide the necessary control for those devices, allowing for a wide variety of individual characteristics such as start/stop times, data formats, and media-handling requirements.
- To achieve these data and control functions with, as nearly as possible, a common set of computer instructions for all I/O devices.
- To synchronize these operations with the internal processes of the computer in a manner which is acceptably economical in its use of both hardware and programming.

- *The 701: Minimizing hardware*

The arrangements in the 701 for handling I/O data were rather spartan, and the stored program in the CPU was involved in the transfer of every word (36 bits in this case) between CRT memory and an input or output device. Several instruction types in the 701 repertoire were provided specifically for I/O operations, and three of these will be described.

The instruction PREPARE TO READ (READ, for short) caused an input device, as specified by the address part of the instruction, to be selected for reading. The selected device could be one of four magnetic-tape drives, or four logical magnetic drums (portions of physical drums), or the single card reader. PREPARE TO WRITE (WRITE) followed similar rules with some rather obvious differences. COPY AND SKIP (COPY) was the instruction which caused each word of input or output data to be transmitted between the multiplier-quotient (MQ) register of the CPU and the location in memory specified by the address part of the instruction. In general, the program had to have a COPY instruction waiting to be executed whenever the input or output device, depending on its timing, required access to a memory location as a destination or source for a word of data.

Depending on the input device (and these same comments apply to output), varying amounts of time between successive COPY instructions were available to the CPU for computing. A COPY instruction, encountered after the last word of the record had been placed in memory, would cause an end-of-record skip of the next two instructions. This skip permitted the program to initiate an instruction sequence appropriate to the end-of-record condition. Similarly, an end-of-file skip of just a single instruction permitted branching to an end-of-file sequence. It can be seen that the necessary synchronization between operation of the CPU and of the I/O system depended to a great extent upon the proper use of the COPY instruction.

The card reader operated in row-by-row fashion, but on only 72 of the 80 columns, so that a card row could be treated as two 36-bit words. The buffering between a card and memory consisted of 72 vacuum-tube flip-flops and the MQ register, all in the CPU, which was a highly CPU-integrated I/O attachment scheme. The card punch and the printer (which electrically resembled a card punch) were attached, and output data supplied to them, in a very similar manner.

The magnetic drums provided data at a rate of one word every 1.28 ms after an initial access time of several milliseconds. Thirty-six tracks were involved simultaneously in the transfer of a word. The sequential data rate of the drums could have been much higher; only one word out of every 128 was read or written. This deliberate slow-down gave the CPU time to perform the calculations needed to deal with the data, including generating subsequent COPY addresses. It also provided time for the regeneration cycles required by the CRT memory of the CPU. One of the most interesting aspects of the drum unit of the 701 is that it was included at all. This was an early, if not the earliest, use of magnetic drums as auxiliary storage in a large-scale computer whose main memory was of a different, faster type.

To summarize, the I/O organization of the 701 was physically simple. It was uncommitted to the transfer of fixed blocks of data or even to dealing with sequential memory locations within an I/O operation. Whenever possible, existing mechanisms were used with a minimum of additional electronics, relying on the CPU program and hardware to carry out housekeeping functions that would later be taken over by such system components as control units and channels.

- *The 702: Simplifying programming*

The I/O capabilities of the 702 illustrate an alternative I/O architecture to that of the 701, in a machine of comparable size and capability. Its planning reflected the assumption that potential users would be most attracted to a computer that dealt with data in the familiar form of existing business records, and one that required the fewest program steps to perform a typical data processing job. An architectural feature which clearly illustrated this orientation was the manner in which READ and WRITE instructions operated. Having previously specified the desired I/O device with a SELECT instruction, the programmer used the address part of the READ or WRITE instruction to designate the location in memory to or from which the first character of an input or output record was to be transferred. Thereafter, as soon as the selected device was available, an entire record was read or written without requiring additional instructions such as the COPY instruction of the 701.

This technique would have left the CPU and memory idle much of the time unless the I/O data rate was high enough. The magnetic-tape data rate, at 15 000 characters per second, came fairly close, being a little over one-third the internal memory speed. The magnetic drum at 25 000 characters per second was a better match, as it used more than half of the available 23-μs memory cycles. For slower I/O units, which included card readers, card punches, and line printers, magnetic-core buffer storage was provided to increase the data-transfer rate and to avoid delaying the CPU while waiting for one of these devices to go through a substantial portion of its mechanical cycle. The rather modest-sized, magnetic-core buffer memory units of the 702, which were delivered starting in February 1955, were among the first coincident-current, magnetic-core storage units produced by IBM.

- *Buffer storage for tape*

As magnetic-tape applications grew in importance, it became clear that keeping the CPU of the 702 busy one-third to one-half of the time was not enough and that some provision for overlapping among tape input, tape output, and processing was needed. At first a small number of magnetic-core buffers were designed and provided for use with 702 magnetic-tape units on a special-engineering basis. Following the 702, the tape buffer idea was kept alive for some years in record storage units made available as regular products for use with the 705. Initially the 705 also provided instructions which overlapped tape reading with writing, but not with processing.

Overlapping of all three was first obtained by means of the Tape Record Coordinator, announced in 1955 for use with the 705. It contained a 1024-character buffer storage, which allowed it to read and simultaneously write a master file on tape while searching for "active" records in the file. Only the active records were passed to the 705 CPU for processing; the rest of the time, the 705 program could proceed independently.

- *Flexibility of attachment*

Right from the start it was considered important to permit a system like the 702 to be tailored to the needs of each installation and to allow for the use of new devices whose characteristics could not be anticipated. Thus any reasonable number of each type of device could be attached, including none. All devices regardless of type were attached via their control units to the CPU and memory by a common cable connection. Circuits peculiar to a particular device type were contained in the appropriate control unit rather than in the processor. This attachment method was the first version of what later became known as a standard I/O interface.

Other systems contributed to the evolving standard interface. The serial input/output adapter of the 1401 permitted attachment of any of (eventually) a wide variety of serial-by-character I/O devices. A major contribution was the Stretch I/O interface, which further generalized the attachability of devices to the system through its "exchange," and included operator consoles, typewriter inquiry stations, and communication terminals in the list of devices attached in this standard fashion.

In the 702 system, the control units for the card reader, punch, and printer were equipped for either direct attachment to the CPU (on-line) or attachment to a tape drive (off-line). Thus the user had a choice of the greater computational efficiency of off-line operation, at the expense of additional delay and tape handling, or the better error control and shorter turnaround time of on-line operation, at the expense of throughput.

The concept of off-line control units underwent an interesting evolution. If a new printer, for example, was equipped with a tape-to-printer control unit, the printer could be used in conjunction with any computer which could furnish data on magnetic tape in a compatible format. This approach avoided both the engineering expense of attaching the printer to the computer and the inefficient use of the computer in feeding data directly to a printer.

But, as mentioned before, when the 1401 was announced in 1959 as a low-end computer system, it was found economical to use the 1401 also as an off-line control unit (with considerable editing capability) to make its superior printing and card-handling equipment available to users of the larger systems of the 7000 series. Thus the 1401 was IBM's first intelligent control unit, an alternative to the "spooling" mentioned in the 7070 description.

• *I/O channels*

As late as 1957, computers were seriously limited in their ability to utilize their processing capabilities during I/O operations, or to perform more than one input or output operation at a time. In the 701 or 704, instructions could be executed during the transfer of a record between memory and the I/O device, but the involvement of the CPU program and the MQ register in the transfer of every word severely restricted the amount of such processing. In the 702, processing was halted during the entire transfer of an I/O record between memory and the device or its buffer storage. Later refinements, such as the Tape Record Coordinator and the read-while-writing feature of the 705, made possible concurrent input and output operations.

Although the 709, one of the last of the vacuum-tube computers announced by IBM, had a rather short technological life, it made an important contribution to concurrent CPU and I/O operations by the introduction of I/O channels [21]. These channels acted individually as I/O processors with a specialized instruction repertoire. Each channel could address and access memory to store or retrieve data quite independently of the CPU program. Coordination between the CPU and channel programs was achieved by CPU instructions for (1) conditional branching depending on whether a channel was in operation, (2) delaying the CPU until completion of a channel command, followed by loading of channel control registers, and (3) storing the channel control register contents in a specified memory location.

For the 709, two channels were packaged in a unit called the data synchronizer, and three such units could be attached to a 709 CPU. Each data synchronizer could handle up to 16 magnetic-tape drives and one card reader, card punch, and printer. With a full complement of six channels, six of these I/O devices could be operating at the same time. (Magnetic-drum and CRT-display units operated independently of the channels.)

It was the parallelism provided by the channels of the 709, as much as any other factor, that led to the development of a supervisory program called the I/O control system (IOCS). It would have been inefficient, when programming each application, to redesign the detailed channel programs and the routines for synchronizing the CPU and channels. The IOCS introduced with the 709 represented one of the early steps in the evolution of the operating system.

Channels similar to those of the 709 were later provided in the 705 III and in most subsequent IBM computers. An equivalent function was introduced through the processing-overlap feature on the 1410, a simplified version of which became available on the 1401. These 1400-series "channels" were integrated into the CPU by means of special registers for holding I/O data and addresses. They required special instructions, but they were not controlled by separate channel programs as in the 700 series and its successors.

All of these channels or equivalent features involved "cycle stealing," a technique for accessing memory in a manner transparent to the CPU program, which had been employed several years earlier in the SAGE computer.

• *I/O interruption*

The instruction loops required on the 709 to test for the status of overlapped I/O operations, and thus synchronize CPU and channel programs, were found to be generally cumbersome and inefficient. A much more direct method is to interrupt the CPU at the end of an I/O operation, or upon the occurrence of exceptional conditions, and to cause the CPU program to branch to an instruction sequence designed to take the desired action. Early versions, such as the data-channel trap of the 7090, were later replaced by more general program-interruption schemes. All of the 7000-series systems, announced between 1958 and 1963, included both I/O channels and program interruption in varying degrees.

The most sophisticated interruption scheme was developed for Stretch, which combined interruptions for I/O and other external causes with the management of program exceptions. It was a direct precursor of the I/O channel and interruption architecture of System/360.

Conclusion

By the end of the fifteen-year period covered by this paper, the computer industry had grown from almost nothing to one of the most important, IBM had changed from being principally involved in punched-card machines to being mainly a supplier of computers, and computers had begun to change the way man dealt with the world.

IBM had several successful lines of computers, but this success also presented a problem. The total effort required to provide systems and application programming for such different computers had become large and could be predicted to become unmanageably so. In the early 1960s, computers were understood much better than in the early 1950s, yet these lines of upward-compatible computers were still constrained to architectural decisions that had been made up to fifteen years earlier. The time was now ripe to replace these multiple older lines of computers with System/360: a group of machines built to a single architecture, that took much better advantage of what had been learned since the beginning of the computer era.

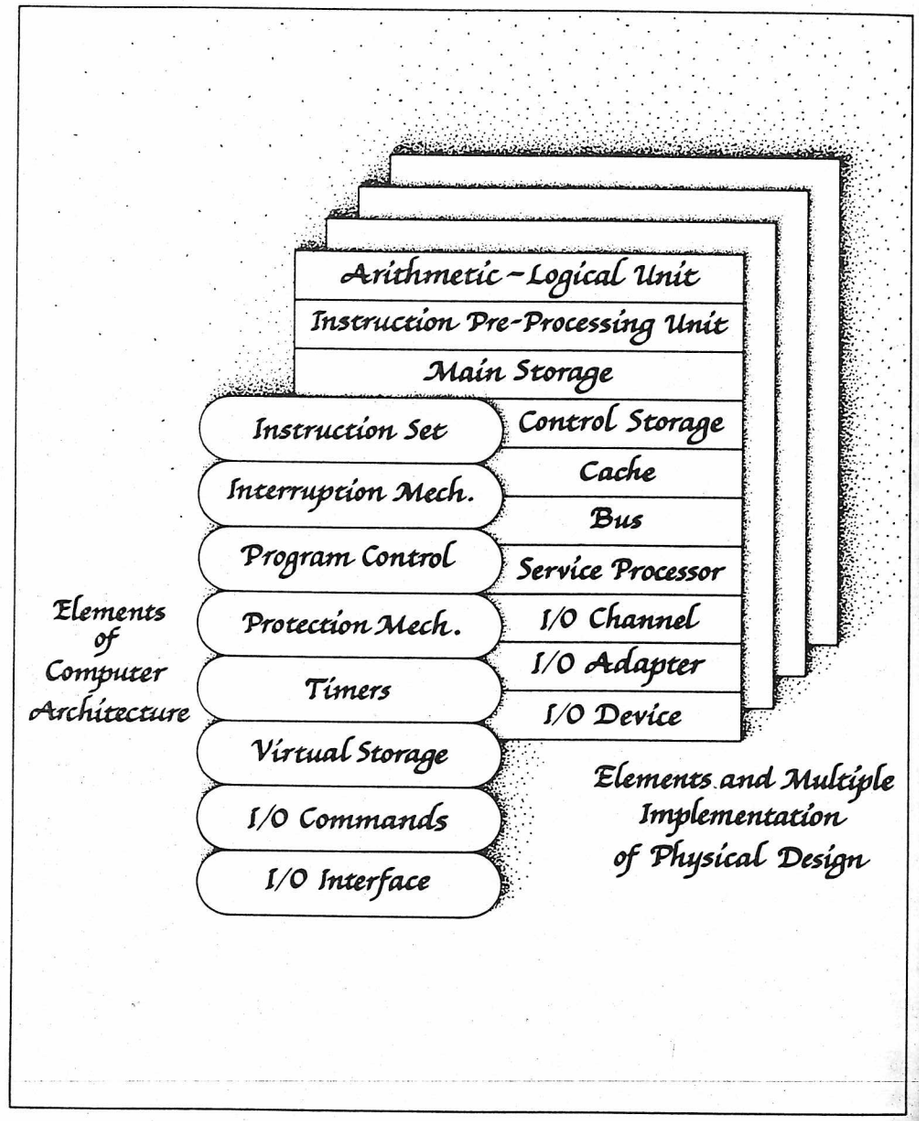
A key factor in finalizing the transition from the older series to System/360 was the advent of emulation. Emulation was a combination of microprogram and software simulation of the various instruction sets, which allowed users to continue to run existing programs efficiently on the new hardware. Thus, at the end of IBM's first fifteen years of developing computers for the marketplace, a new evolution was starting, one which has not ended to this day.

References

1. For additional information concerning precursor machines, see Byron E. Phelps, "Early Electronic Computer Developments at IBM," *Annals of the History of Computing* **2**, 253–267 (1980).
2. L. D. Stevens, "Engineering Organization of Input and Output for the IBM 701 Electronic Data Processing Machine," *Proceedings*, Joint AIEE-IRE-ACM Computer Conference, New York, December 1952, pp. 81–85.
3. M. M. Astrahan and N. Rochester, "The Logical Organization of the New IBM Scientific Calculator," *Proceedings of the Electronic Computer Symposium* (IRE), Los Angeles, April 30–May 2, 1952, paper XIX (7 pp.).
4. W. Buchholz, "The System Design of the IBM Type 701 Computer," *Proc. IRE* **41**, 1262–1275 (1953).
5. C. E. Frizzell, "Engineering Description of the IBM Type 701 Computer," *Proc. IRE* **41**, 1275–1287 (1953).
6. J. L. Greenstadt, "The IBM 709 Computer," *Proceedings of the Symposium: New Computers, a Report from the Manufacturers* (ACM), Los Angeles, March 1957, pp. 92–96.
7. N. Rochester, C. J. Bashe, W. Buchholz, R. P. Crago, P. E. Fox, J. A. Haddad, and B. E. Phelps, "Electronic Data Processing Machine," U.S. Patent 3,245,039, April 5, 1966 (448 pp.).
8. C. J. Bashe, W. Buchholz, and N. Rochester, "The IBM 702, an Electronic Data Processing Machine for Business," *J. ACM* **1**, 149–169 (1954).
9. C. J. Bashe, P. W. Jackson, H. A. Mussell, and W. D. Winger, "The Design of the IBM Type 702 System," paper no. 55-719, *AIEE Transactions* **74** (Part I, Communication and Electronics), 695–704 (1956).
10. E. S. Hughes, Jr., "The IBM Magnetic Drum Calculator Type 650, Engineering and Design Considerations," *Proceedings of the Western Computer Conference*, Los Angeles, February 1954, pp. 140–154.
11. F. E. Hamilton and E. C. Kubie, "The IBM Magnetic Drum Calculator Type 650," *J. ACM* **1**, 13–20 (1954).
12. M. L. Lesser and J. W. Haanstra, "The Random-Access Memory Accounting Machine—I. System Organization of the IBM 305," *IBM J. Res. Develop.* **1**, 62–71 (1957).
13. R. W. Avery, S. H. Blackford, and J. A. McDonnell, "The IBM 7070 Data Processing System," *Proceedings of the Eastern Joint Computer Conference*, Philadelphia, December 1958, pp. 165–168.
14. J. Svigals, "IBM 7070 Data Processing System," *Proceedings of the Western Joint Computer Conference*, San Francisco, March 1959, pp. 222–231.
15. R. R. Bender, D. T. Doody, and P. N. Stoughton, "A Description of the IBM 7074 System," *Proceedings of the Eastern Joint Computer Conference*, New York, December 1960, pp. 161–171.
16. W. Buchholz, Ed., *Planning a Computer System (Project Stretch)*, McGraw-Hill Book Co., Inc., New York, 1962.
17. S. S. Snyder, "Computer Advances Pioneered by Cryptologic Organizations," *Annals of the History of Computing* **2**, 60–70 (1980).
18. W. J. Eckert and R. Jones, *Faster, Faster*, McGraw-Hill Book Co., Inc., New York, 1955.
19. R. R. Everett, C. A. Zraket, and H. D. Benington, "SAGE—A Data-Processing System for Air Defense," *Proceedings of the Eastern Joint Computer Conference*, Washington, DC, 1957, pp. 148–155.
20. M. M. Astrahan, B. Housman, J. F. Jacobs, R. P. Mayer, and W. H. Thomas, "Logical Design of the Digital Computer for the SAGE System," *IBM J. Res. Develop.* **1**, 76–83 (1957).
21. C. L. Christiansen, L. E. Kanter, and G. R. Monroe, "Data Synchronizer," U.S. Patent 3,812,475, May 21, 1974.

Received April 11, 1980; revised September 25, 1980

C. J. Bashe is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. W. Buchholz is with the Data Processing Products Group at the IBM laboratory in Poughkeepsie, New York 12602. G. V. Hawkins is located at 445 Hamilton Avenue, White Plains, New York 10601. J. J. Ingram is located at the IBM System Communications Division laboratory, Research Triangle Park, North Carolina 27709. N. Rochester is located at the IBM Cambridge Scientific Center, 545 Technology Square, Cambridge, Massachusetts 02139.



Computer architecture and its implementation

A. Padegs

System/360 and Beyond

The evolution of modern large-scale computer architecture within IBM is described, starting with the announcement of System/360 in 1964 and covering the latest extensions to System/370. Emphasis is placed on key attributes and on the motivation for providing them, and an assessment is made of the experience gained in the implementation and use of the architecture. The main approaches are discussed for obtaining implementations at widely differing performance levels, and a number of significant implementation parameters for all processors are listed.

Introduction

With the introduction of System/360 in 1964, a major change in the development of computers within IBM took place. With the recognition that architecture [1] and implementation could be separated and that one need not imply the other, a common machine architecture was established. It was intended for program-compatible embodiments over a wide range of performance levels and for various types of applications.

Since its introduction, this architecture has been the basis for all intermediate and large computers produced by IBM. It has also become the basis for machines produced by a number of other manufacturers in the USA, Japan, and the Soviet Union. The architecture has provided a firm interface for application development, and it has permitted the operating systems to grow significantly in size and function. Although the architecture was developed when logic technology with a single device per chip and magnetic cores were used to implement machines, its fundamental structure is still suitable for today's designs, which use dense arrays and integrated circuits of thousands of elements per chip.

This paper reviews the salient characteristics of the System/360 architecture and its follow-on, the System/370 architecture. The objectives are to cover significant accomplishments, to give the motivation for key architectural decisions, and to present an assessment; it is not intended that the paper provide a complete historical record of IBM's architecture developments. Furthermore, the paper does not contain a detailed technical review of design considerations and alternatives; this type of review has been published previously [2-9].

The first two sections review the System/360 and System/370 architectures, stating the objectives, constraints, and contributions. Following this, developments in I/O architecture are outlined. The introduction of various enhancements to the architecture is discussed in relation to product announcements; and, in a separate section, the significance of microprogramming and the cache to the implementation of a compatible line of machines is reviewed. The final sections are devoted to an assessment of IBM's experience with System/360 and System/370. In an appendix, tables comparing implementation characteristics for all processors provide further illustration of the steps taken to achieve different performance levels.

System/360 architecture

System/360 was the result of a major effort to design an architecture for a new line of computers that was unencumbered by the requirement to be compatible with existing architectures. Work on a new architecture for a family of machines began in the early 1960s; its specifications were released in April 1964, when the first models of System/360 were announced.

When System/360 development was initiated, most new computer models were, from the viewpoint of their logical structure, improved, enlarged, or technologically recast versions of the machines developed in the early

1950s. IBM products had evolved from 701 to 7094 II, from 702 to 7080, from 650 to 7074, and from 1401 to 7010 [10]. Additionally, IBM had produced Stretch, formally known as the IBM 7030; it had been developed largely as a project to challenge the state of the art, but from the point of view of architecture it was a predecessor of System/360.

In many ways the design concepts underlying System/360 [2–5] were the same as those for Stretch [11]. Both Stretch and System/360 provided, in a single architecture, facilities suitable for scientific, commercial, and real-time applications; both placed major emphasis on the generality and code-independence of instruction and data formats; and both provided for the uniform attachment and control of a wide variety of I/O equipment. System/360, however, was the first demonstration that the concept of developing an architecture for a family of compatible machines was practical, with the initial implementations targeted to yield models with internal performances ranging from that of the IBM 1401 to well beyond that of Stretch. Intermodel compatibility was probably the most far-reaching requirement in developing System/360 and one which affected both the architecture and the procedures for developing and controlling it.

System/360 incorporated a number of the new architecture concepts introduced in Stretch, such as binary storage subdivision and the eight-bit byte, storage protection, and a generalized interruption mechanism. However, the System/360 architectural definition of some of these concepts differed from that in Stretch because the environment and objectives were different. Since Stretch had the flavor of an experimental computer, its architecture could afford to strive for a set of functions of great logical consistency and completeness. System/360, on the other hand, was defined at its inception as a base for a product line. As such, its development called for a more frugal choice in the selection of functions, based on a critical evaluation of the available experience. The architecture had to encompass implementations covering wide performance and cost ranges, and its definition had to reflect compromises between the performance and cost objectives of large and small machines. As a result, Stretch innovations such as storage addressing to the bit level, variable byte size, and automatic handling of floating-point range exceptions were not included.

The following are the key areas where System/360 introduced innovations or otherwise determined direction in architecture.

Addressing For efficiency and because of the ease of table utilization, System/360 uses binary-radix storage addressing, with 24-bit addresses that designate byte locations. Thus System/360 can address 16M eight-bit bytes, as compared to the 2M bytes on Stretch (M stands for 2^{20} or 1 048 576). This addressing capability is, of course, available on all models and should be considered in light of the maximum storage sizes available at that time: 32K 36-bit words on the 7094 II (K stands for 2^{10} or 1024), 160 000 six-bit characters on the 7080, 300 000 digits on the 7074, 100 000 seven-bit characters on the 7010, and 16 000 seven-bit characters on the 1460 (the word-mark bit is included in the 7010 and 1460 character sizes). The smallest initial System/360 model, Model 30, offered up to 64K eight-bit bytes, and 1M bytes was available on the largest initial model, Model 75. The ability to address and to effectively utilize large storage was one of the key attributes of System/360.

Address generation A truncated 12-bit address in an instruction, in conjunction with a full base-address value in a register, provides indexing and eliminates the inefficiency of carrying a full address in each instruction. A second level of indexing is available in some instructions to facilitate loop control. The 12-bit displacement, without an index or base value, provides addressability for loading or saving the base values but normally is so small that all programs need base addresses and are thus generally location-independent.

Provision for control program A comprehensive interruption system, supervisor and problem states, storage protection, and an interval timer provide a basis for designing a secure operating system. Input/output instructions are invalid in the problem state, and means are provided for the supervisor to control the duration of application programs and switching between them. (In Stretch an operating system could be modified by unauthorized input from an I/O device.)

Input and output The multiplexer channel and a common method of attaching and programming all I/O devices extended the concepts introduced in 702 and Stretch. These aspects are discussed subsequently in the section "Input and Output."

General-purpose registers Sixteen registers serve as accumulators for fixed-point and logical operations, as well as sources of base and index values in address generation, thus bringing the full power of the fixed-point arithmetic-operation set to bear upon indexing computations.

Character size In contrast to the straight six-bit approach used in the IBM 702–7080 and 1401–7010 families, two character sizes were introduced: eight-bit codes for alphanumeric, and four-bit codes for numeric characters. This approach, used in the IBM 650–7074 family, has

greater coding efficiency, with spare code points in the alphabetic set, and is commensurate with binary subdivisions used in the rest of the system. The length of operands is specified in the instruction: Decimal operands can be up to 16 bytes in length; character operands are variable up to 256 bytes. In Stretch any character size from one to eight bits could be specified, but variable-field-length operands were limited to 64 bits.

Floating-point data format Two formats were introduced, both available on all models: a 64-bit format for use in precision-sensitive problems and a 32-bit format for faster speed and conservation of storage space. The 32-bit format was intended primarily for the smaller models, where differences in the execution time for the two formats were significant. The alternative would have been a single 48-bit format to succeed the 36-bit format of the 7094 and the 64-bit format of Stretch. Both the 32-bit and 64-bit formats use the same exponent size, with a base of 16. This was a departure from base 2 and was introduced to permit simpler circuitry and to reduce the frequency of pre-shift, overflow, and precision-loss post-shift in addition and subtraction [12].

Serviceability The ability to automatically record the detailed machine state at the instant of an error and to initialize it to any specified value provides tools for significant serviceability improvements [13].

The models at both ends of the performance range introduced architecture changes to meet their particular cost and performance goals. Model 20, although nominally called part of the System/360 family, was incompatible with System/360. Its architecture provided for a maximum main storage of 64K bytes, and it had 37 instead of System/360's 143 instructions. Other differences were that it did not include the supervisor state, had its own set of I/O instructions, and had a 32-bit instead of the 64-bit program-status word (PSW). Compatibility for running application programs was affected because the Model 20 omitted floating-point and 32-bit binary arithmetic, had eight 16-bit instead of sixteen 32-bit general registers, and had a special direct-addressing mode for forming storage-operand addresses.

At the other end of the performance range, the Models 91, 95, and 195 introduced some deviations to accommodate their highly overlapped designs by delaying program interruptions and permitting the result of a divide operation to be off by one bit in the low-order bit position. These differences required some adjustments in software but did not affect compatibility for application programs in any significant way.

Additional functions were introduced by a few of the later System/360 models. Model 44, announced in 1965, had a number of extensions for real-time applications, which, however, were not continued in later models. At the same time, Model 67 introduced virtual storage [14], which, with some modifications, became a basic part of System/370. The 9020 System, which was developed for the Federal Aviation Administration, interconnected modified Model 50s into a multiprocessing system to meet exceptionally stringent requirements for continuity in machine operations [15]. Models 65 and 67 both offered multiprocessing facilities [6], which later were significantly extended for System/370.

Then, in 1968, as part of the extended-precision floating-point facility on the Model 85, the 128-bit floating-point format was introduced [7]. The package also included special instructions for rounding floating-point numbers when going to a shorter format; they alleviated somewhat the lack of rounding in the original architecture. Model 85 also removed the original System/360 requirement that storage operands for unprivileged instructions be aligned on boundaries equal to a multiple of the operand length. Both of these extensions were carried into System/370. Additionally, the 2880 Block-Multiplexer Channel, on System/360 Models 85 and 195, had many of the System/370 I/O architecture extensions [8].

In two areas the original System/360 decisions on user-oriented functions were subsequently changed. The first one concerns floating-point instructions. The original System/360 architecture did not anticipate the significance of compatibility in the handling of overflow and the need for indicating the true result value on both overflow and underflow. Furthermore, it had overlooked the need for a guard digit in post-normalization. Both of these functions were changed in 1968, with all installed machines retrofitted.

The other change concerns the encoding of decimal data. System/360 anticipated the adoption of a proposal for a seven-bit American Standard Code for Information Interchange (ASCII) and of a technique for expanding the seven-bit code to eight bits. It provided a mode bit in the PSW that specified, for the code-sensitive instructions, operation with either the ASCII or the Extended-Binary-Coded-Decimal-Interchange Code (EBCDIC). The eight-bit extension of the code was never adopted as a national standard, and the ASCII mode has subsequently been deleted in the System/370 architecture. It was highly unlikely that any production programs ever used the eight-bit ASCII code; none have ever been identified. The mode bit subsequently turned out to be the only unassigned one in the PSW and was convenient for distinguishing between the original and the extended-control (EC)-mode PSW format introduced for System/370.

In a number of other areas, different architectural choices, in retrospect, might have been preferable. The problem of storage-address size is discussed later in the paper. For some areas, extensions have subsequently been introduced to solve the constraints set by the initial decisions: for example, the new EC-mode PSW format corrected the lack of extensibility in the original PSW format, and the early release of the CPU during execution of the new System/370 START I/O FAST RELEASE instruction eliminated the unnecessary delay required by the original START I/O in high-performance machines. In other areas, the need to preserve compatibility has been felt to be so overwhelming that the reevaluation of the original architectural choices has been of no practical interest. This applies particularly to problem-program functions, such as whether the floating-point significance exception is really useful, and whether the EDIT and EDIT AND MARK instructions are warranted in view of their very specific operand formats.

System/370 architecture

In contrast to System/360, the objective of System/370 was an evolutionary extension of System/360 architecture for a new set of models and for new releases of programming systems [9]. Experience with the System/360 architecture had identified a number of bottlenecks and limitations in the efficiency of system use and had pointed out areas where additional machine functions were desirable. Furthermore, because the cost of technology for main storage and logic circuitry was becoming lower in relation to the overall system cost, it was feasible to consider extending the machine architecture; it was possible, in fact, to economically include functions that did not appear justified when System/360 was developed. For example, because of cost considerations in the smaller models, System/360 provided only one 32-bit timer in a main-storage location, which had to be programmed to provide all timing functions. System/370 introduced three distinct facilities, each with a 64-bit value: a time-of-day clock (for real-time indication), a clock comparator (a real-time alarm clock), and a CPU timer (for measuring process time) [9].

System/370 was constrained to be upward-compatible for System/360 application programs and for the main-line operating systems. Even though such operating systems could not benefit from the new functions available in System/370, and new support was planned, the ability to run those operating systems was needed for the transition period. For this reason, System/370 continued to provide functions, such as the System/360 timer and the System/360 PSW format, that had in fact been superseded by functionally richer extensions. Additionally, System/370 needed to attach and operate System/360 I/O devices.

System/370 evolved, and its architecture [16] was released, in a number of increments. The system was introduced in June 1970 with the announcement of Models 155 and 165, at which time the main architectural extensions were six general-purpose instructions, the time-of-day clock (with a period of 143 years and a resolution of one microsecond), and control registers (they serve as an extension of the PSW). The original System/370 also included a number of extensions to enhance model-independent recovery by software from machine malfunctions [17]. Virtual storage, the CPU timer, the clock comparator, program-event recording (for software debugging), and the new PSW format and interruption controls associated with the extended-control (EC) mode were introduced with the announcement of Models 158 and 168 in August 1972. Multiprocessing and the conditional-swapping and PSW-key-handling instructions were introduced in February 1973.

Then, with the introduction of the 3033, a number of extensions were made available that enhanced the performance and function of the MVS operating system. One-level addressing and the instruction MOVE INVERSE were introduced as part of the VSE (virtual storage extended) mode on the 4300 processors to meet the needs of the DOS/VSE operating system [18, 19]. The MOVE INVERSE instruction is intended primarily for environments, such as the Arabic language, where text is arranged in a right-to-left order.

The single item that most distinguishes System/370 from System/360 is the availability of a dynamic-address-translation facility, which allows the control program to efficiently implement a group of functions collectively referred to as *virtual storage*. The approach incorporates *paging* from external storage as introduced in Atlas [20] and a second level of indirection, *segmentation*, as suggested by Dennis [21] and as further detailed by Arden *et al.* [22].

The System/370 version of this facility is largely patterned after the System/360 Model 67 [14]. Experience with that machine and its operating system, TSS, had verified the value of many of its concepts and had provided actual usage data for making System/370 design decisions. In addition to a number of format changes, System/370 offers two page and segment sizes to accommodate both large and small systems, but it does not offer 32-bit virtual addressing, which was available on the Model 67. The System/370 virtual-storage operating systems were evolutions of the corresponding real-storage operating systems and could not accommodate 32-bit addresses.

The virtual-storage architecture of the 3033 and other large processors was enhanced early in 1981 by the introduction of the dual-address-space facility. This extension includes a 16-bit address-space number, which is associated with a set of segment and page tables and identifies a virtual address space of 2^{24} bytes. A total of 2^{16} address spaces can be established, although at any one time addressability exists to two address spaces—the primary and secondary. Instructions and controls are provided to load an address-space number so as to establish addressability, call and return from programs in either the same or another space, move data between spaces, and establish authorization for these operations. These facilities extend the size of the addressable virtual storage and provide a basis for enhancing system integrity.

In the VSE mode, the main change was the substitution of the *one-level-addressing* facility for the System/370 dynamic-address-translation facility. DOS/VSE offers one virtual address space of up to 16M bytes, and the architecture is simplified accordingly by eliminating the multiple-address-space capability of the System/370. Storage is directly addressable by the CPU and all channels, using a uniform set of virtual addresses. The translation table is in internal machine storage, and special instructions are provided for setting up the mapping. Protection, by means of storage keys, applies to virtual instead of real storage. Because of the simpler translation procedure and the ability of channels to use virtual addresses, performance gains are possible, and the software for translating addresses in channel programs is eliminated. The VSE mode is compatible with System/370 for problem programs, but not for the control program. The full System/370 facilities are available on the 4300 processors in the System/370 mode.

Another major functional extension is the inclusion of a number of facilities that permit formation of a *multiprocessing* system, where two or more CPUs share common main storage and are controlled by a single copy of the operating system. The concept of using a prefix to offset the main-storage address when accessing the block containing shared control information is the same as that used in System/360. The architecture was extended, however, by making the prefix settable by the program (instead of manually) and by providing the SIGNAL PROCESSOR instruction and a special interface for communicating between CPUs. On the 3033 a further extension made it possible for the software to connect a set of channels to one of two CPUs.

The main extension to the multiprocessing architecture, though, was in the control of accesses to shared main storage. In a multiprocessing system, the conventions of a uniprocessor communication protocol become inadequate when one CPU is changing the contents of a common storage location while the other is observing it, or when both CPUs are updating the contents of the location at the same time. The System/370 architecture includes a number of rules on the concurrency, multiplicity, and order of storage accesses, and specific instructions are introduced to permit sharing of serially reusable resources, such as updating chained lists. Specifically, in System/360, the TEST AND SET instruction provided a means whereby the inspection of a bit in storage and the setting of it to one could be performed indivisibly. In System/370, the two compare-and-swap instructions indivisibly compare a field in storage with a value in a register and, upon matching, replace the storage operand with a new value [16].

The *IBM System/370 Principles of Operation* [16] in the Spring 1981 edition contained a total of 204 instructions, as compared to the 143 initially available in System/360 [23]; this provides one indication of the growth of the architecture. Of the 61 new instructions, 39 are either privileged or semiprivileged (11 of the original System/360 instructions were privileged), indicating that a relatively larger portion of the architectural extensions is intended for system functions.

Input and output

The concept of a common method of I/O attachment and control evolved gradually. The 702 had a common interface for attaching I/O control units and a common architecture for controlling I/O operations. The 709 introduced the concept of a channel. The Stretch "exchange" [10] provided a mechanism for sharing equipment for multiple I/O operations, using a common I/O interface and I/O architecture (a different interface was used for attaching the disk unit to the high-speed exchange, and the instructions for its control differed somewhat). In 1961 a standard interface was established for attaching I/O to all new large systems. It was a modification of the Stretch interface and was available on the IBM 1410/7010, 7040/44, 7070/74, 7080, and 7090/7094 systems for attaching disk, magnetic tape, and communications control units.

System/360 extended the standardization of I/O attachment and control by applying a common attachment interface and a uniform program control to a larger variety of device types and covering a wider spread of data rates.

- *Channels*

System/360 introduced the *subchannel*, which for most purposes gives the appearance of an independently oper-

ating processor that can sustain its own channel program. Different types of channels were designed, and, depending on the type of channel, different levels of concurrency among channel programs were made possible. A *selector* channel has one subchannel and permits operation with one device at a time, normally at a high data rate. A *multiplexer* channel can have up to 256 subchannels and, conceptually, can be executing a channel program for each subchannel. The actual level of interleaving depends on the type of multiplexer channel and the device. A *byte-multiplexer* channel is designed for low-speed operation to interleave individual bytes or bursts of bytes from such devices as keyboards, communications lines, printers, and card equipment. When it was introduced, it represented a major advance for communications-based systems and real-time applications [5]. The *block-multiplexer* channel, introduced later for System/360 Model 85, is intended for high-speed operation and is particularly advantageous for use with rotating storage devices, such as disks and drums [8]. When used in conjunction with rotational-position sensing, it permits a subchannel to be assigned and a channel program to be established for each access arm, with each program monopolizing the channel for the duration of data transfer but releasing channel facilities during arm movement and during the rotational delay associated with locating the designated record.

- *I/O interface*

The *System/360 I/O interface* is the connection between a channel and an I/O control unit; it provides the necessary physical, electrical, and communications-protocol specifications. It is based on the standard interface of 1961.

The original System/360 I/O interface specification was adequate for data rates up to about 1M bytes per second for a cable length of about 100 feet. For cable length of the order of 20 feet, the IBM 2301 Drum Storage, with a rate of 1.2M bytes per second, could be accommodated. The fully interlocked signaling protocol allowed one channel-cable connection to sustain data transfer over a very wide range of rates, with both the channel and device having complete control of the timing of each byte transfer. It did, however, require an electrical signal to be propagated between the channel and the control unit four times for each byte transferred.

With the advent of auxiliary-storage technologies employing higher recording densities, it was necessary to increase the data-transfer capacity of the interface. For some buffered devices a higher data rate was desirable to reduce the transfer time. Furthermore, many installations needed longer cable connections. To meet these goals, System/370 introduced changes both in the signaling protocol and in the width of the interface.

The System/370 I/O interface [24] includes two additional tag lines to provide the same level of transfer interlocks with only two propagation times per byte transferred. It depends on the control unit whether or not the new facility is used; thus, control units implemented to operate with the System/360 protocols can be attached to System/370 channels. On some System/370 channels and control units the bus width can be extended optionally to two bytes, thus doubling its data-transfer capacity.

As a result of these two additions, the System/370 I/O interface can sustain a data-transfer rate of over 1.5M bytes per second in the one-byte version and over 3M bytes per second in the two-byte version, over a cable length somewhat less than 100 feet; longer distances can be accommodated at lower data rates.

The *data-streaming* mode, introduced recently for the IBM 3380 Disk Storage, eliminates the interlocks between the request and response signals during data transfer. Data, with the appropriate tag signals, are sent in the form of fixed-length pulses. This eliminates the dependency of the data rate on cable length caused by the interface protocol. The IBM 3380 specifications provide for a transfer rate of 3M bytes per second over 400 feet with the one-byte interface.

System/360 was the first system in which a common attachment interface was used to connect a large variety of I/O control units to a line of computers. The interface has been successful in a number of ways. It has offered an unprecedented choice of I/O equipment in configuring a system. It has permitted channels and control units to be designed independently and at different locations with an assurance that, assembled into a system on the user's premises, they will operate without any adjustments. Furthermore, the specific interface definition has been sufficiently general and flexible to accommodate new device types and to permit extension of function and data rates in a compatible manner. As a result, a control unit designed to the original definition (after the 1967 change to the electrical specifications) can operate with a channel incorporating the latest extensions, provided the channel meets the speed requirements.

The standard interface permits other attachment approaches. At the penalty of losing some configuration flexibility, the total cost of a system can be reduced by eliminating a separate frame and power supply for the control unit, by eliminating the use of an interface cable and the associated drivers and receivers, and by sharing some main-frame logic circuits for the control-unit functions. Such integrated designs are offered on the smaller System/360 and System/370 models for some common I/O device types. Even though such designs physically merge the channel and control unit, they nevertheless maintain the logical separation and simulate those aspects of the standard interface that are observable by the program. Thus, regardless of the implementation, all I/O devices are controlled by the same set of I/O instructions, command words, and other program formats.

Implementation approaches

The various levels of performance and cost in the implementation of the architecture are achieved by appropriate choices and tradeoffs among such parameters as circuit speed and cycle time, width of data and logic paths, overlap of instruction execution, and speed, width, and interleaving of main storage [25, 26]. Two new developments in machine implementation, however, are particularly significant in the adoption and subsequent extension of System/360 architecture: microprogramming and the cache (high-speed buffer).

- *Microprogramming*

Microprogramming, originally suggested by Wilkes [27], is the use of simple and fast low-level instructions for controlling machine sequences [28–30]. This type of design permits sharing a basic data flow for a wide variety of functions and readily permits tradeoffs between cost and performance. With conventional logic circuitry, the cost of controls increases in a roughly linear relationship to the functional capability. With microprogramming, a base cost for the microcode-storage device and the supporting logic must be borne, after which the incremental cost for adding more storage in order to microprogram additional function is relatively small.

It was largely because of microprogramming and the economy associated with sharing hardware that it became economically feasible to implement the full System/360 architecture on the smaller models. The savings were particularly significant in the implementation of input and output, as microprogramming made it possible to build integrated channels where the logic capability of the machine is time-shared between CPU and channel functions. In such an implementation, the channel becomes a conceptual entity, and one may include a large number of subchannels at virtually no cost other than the storage space for the governing control information.

Microprogramming made it possible to incorporate in System/360 and System/370 models the capabilities for emulating other architectures, such as those of the IBM 1401 and the IBM 7094 [31]. It also made it possible to extend the original System/360 architecture with assists for specific operating systems. Furthermore, microprogramming has had a beneficial effect on the architecture-resolution process, since it permits corrections and changes in machine functions after the circuitry has been designed and built; some changes are feasible even after the machine has been delivered to the customer.

Microprogramming is used to varying extent in all System/360 and System/370 models except for Models 44, 75, 91, 95, and 195. The extent and the method of use depend on performance objectives. Larger models normally have more bits per microprogram-instruction word for the control of their more complex data paths. On the other hand, smaller models have larger microprograms, since these models require more cycles to accomplish the same function and use microprogramming for more functions. As an example, the Model 168 has 4K words of 108 bits each, whereas the Model 138 has 64K words of 18 bits each. In the initial System/360 models, microprograms resided in read-only storage, but in most later models read-write storage is used. In the smaller models, microprograms reside in an extension of main storage.

- *Cache*

Starting with System/360 Model 85, the larger models use a high-speed buffer, called the *cache*, for accesses to main storage. Although the concept had been considered previously [32], IBM was the first to implement a large cache in a commercial computer [33–36]. The cache was a major advance in system organization and subsequently has been extensively analyzed in the literature [37–39]. The cache is interposed between the CPU and main storage, and its existence is not apparent to the program.

The cache reduces the number of main-storage references, because information fetched into the cache can be reused without access to main storage. Furthermore, by loading entire "lines" (typically 32–64 bytes) on any request for storage information, the machine can prefetch valuable information for future use and thus avoid the delay associated with additional storage access. The effectiveness of the cache depends on its size and other design parameters, as well as on the distribution of addresses used to access storage. According to Liptay [34], on the Model 85 with a 16K-byte cache, typically 97% of fetches were satisfied with data from the cache. With larger caches, in scientific applications "hit" ratios of 99% and over can be attained, although for interactive environments a more typical ratio is 96%. Furthermore, by allowing channels to communicate directly with main storage, the cache reduces storage interference and improves accessibility of storage for I/O, thus permitting higher I/O data rates.

The effect of a storage hierarchy using a cache is to reduce the dependence of CPU operations on storage access time and to provide a better match between the

operation speeds of main storage and CPU circuitry. The cache provides more freedom in the choice of storage technologies and allows for larger storage and longer access times. The introduction of a cache played a significant role in the realization of systems with large main storage.

Experience with System/360 and its extensions

The following are some of the major observations to be made and conclusions to be drawn concerning System/360 and its extensions.

• *Implementation of compatible machines*

Experience clearly verified that the initial System/360 goals for a compatible line of machines were realistic, and that it was feasible to build a family of machines within which programs could be transferred routinely from one model to another. The validity of the original compatibility goals was particularly proven by the fact that other manufacturers have been successful in producing System/370-compatible machines. In fact, compatibility helped reduce development costs within IBM. The original System/360 plan called for verifying each element of software on each model. Because of the growing confidence that programs which ran on one model would also run on other models, it was possible to significantly reduce the amount of cross-verification performed.

The original System/360 announcement included processors with a performance range of 25 to 1. Six years later this had increased to around 200 to 1, and today the performance of the 3081 is approximately 450 times that of the System/360 Model 25.

• *Main storage*

Main-storage sizes grew more rapidly than was anticipated in the 1960s; the technological improvements, which reduced the cost, had occurred at a faster rate than was expected. Thus, it became obvious at the time System/370 was in the planning stages that the 24-bit main-storage address size would have to be extended eventually.

The extension of the address size, however, proved to be more difficult than first expected. The basic addressing mechanism of System/360 was well suited to extension, since it depended on base registers that were already 32 bits wide. The interruption mechanism and the I/O control formats, however, did not have the required extensibility, since immediate cost and performance consequences in 1962 had outweighed the need to meet eventual long-term requirements. More importantly, operating systems and compiler-produced application programs had used the extra bit positions in address words for control purposes and hence required extensive modification.

In all new formats introduced for System/370, such as the control registers and the EC-mode PSW, main-storage-address fields are assigned 32 bit positions, should they be needed for address expansion. On the 3033 and other large machines, however, real storage in excess of 16M bytes is accommodated by making use of unused bit positions in the translation tables.

• *Precision vs. unpredictability*

In order to ensure compatible implementations, the architecture has to be complete in that it must cover all functions of the machine that are observable by the program, including all the unlikely concurrent occurrences of different unusual exceptions. It either must specify the action the machine performs or state that the action is unpredictable.

Identical action in all machines is less likely to cause problems with compatibility and has a certain aesthetic appeal. Indiscriminately specifying predictable operation, however, may present problems when the predictable operation is of insignificant value to the user and some later machine has difficulty complying with the required predictability. Whereas specifying initially that an operation is unpredictable might have been quite acceptable, relaxing the architecture definition to permit unpredictability has certain risks, because some programs may have come to depend on the initial, precise definition. Thus the architect has to make a deliberate decision about the extent of predictability.

The System/360 architecture did not provide adequate precision and detail in some areas. Because there was no specification of the priority in which concurrently existing program exceptions are recognized, programming of virtual machines was made difficult. Because the sequence and concurrency for storage accesses were not specified, processors could not communicate reliably using shared main storage. And because not enough details in machine-check handling were specified, the possibility of model-independent recovery after an equipment failure was reduced. The 1973 edition of the System/370 definition was more detailed and precise, but, for the sake of simplicity of the architectural model, specified as predictable some aspects that, as experience indicated, should not have been. An appendix in the 1980 edition of the *System/370 Principles of Operation* [16] lists six changes where the requirements for predictability have been relaxed. These changes concern such aspects as indicating an access exception for an operand when the instruction can be completed without the use of the operand, and they are unlikely to affect any program.

• *Assists*

In addition to the general-purpose architecture included in the *Principles of Operation*, many CPUs include special-purpose functions to improve the performance of a specific programming system. These functions, referred to as *assists*, comprise frequently occurring instruction sequences of a particular application, and a single operation code (or the occurrence of some other condition) may invoke the execution of an extensive procedure.

The assists are made possible by microprogramming and are implemented mostly (and in most machines exclusively) in microcode. They are particularly effective in improving performance when the function includes an interruption sequence and the associated program action; for example, when operating under VM/370, depending on the model and the operating system, a 40–65% reduction in elapsed time due to the VM assist has been measured [40].

The assists, however, are temporary internal interfaces and are not intended for application-program development. The functions may change between releases of the operating system, and, since the design decisions may be made on the basis of tradeoffs involving only a few specific machines, they may vary between models.

• *Levels of compatibility*

With the establishment of the operating system as an essential component of a user's installation, part of the architected machine interface is becoming an internal interface between the machine and the operating system. The dynamic-address-translation mechanism and machine-check indications are some examples of functions that do not directly affect the user, but the operating-system-dependent nature of the interface is particularly emphasized by the introduction of the assists. Furthermore, since the larger models normally are used with functionally richer operating systems and since the smaller models are usually restricted to those with lower storage requirements, an affinity has developed between machine power and operating-system power. Because of the nature of this affinity, it is not essential that the part of the machine interface affecting only the operating system be the same on all machines.

This evolution points out that two types of requirements for compatibility have to be considered. In order that old application programs run on new machines, the machine, jointly with the system program, must ensure that the basic facilities intended for *application-program* development continue to be available. On the other hand, changes may be acceptable in those facilities that are available to and affect only a *system program* or that can be masked by the system program from the application program. Indeed, such changes have to be expected, since they make it possible to improve the performance of system functions.

Such changes (as contrasted to extensions) have been introduced at different times into the System/360 and System/370 architectures. In the VSE mode on the 4300 processors, the one-level-addressing facility replaced the dynamic-address-translation facility in order to improve virtual-storage management for small systems; it affected only the interface between the machine and the DOS/VSE operating system that uses it. Similarly, it was feasible to phase in the extended-control (EC) mode, with the associated changes in interruption control and the PSW format, since the machine format affected only parts of the control program. The OS/VS2 operating system, however, continued for years to maintain the original PSW format in areas where the format was exposed to the user, such as in the trace information.

Architecture control

The design of a compatible line of machines required a strict separation of the architecture and machine-design functions and the introduction of methodology for the control of architecture. One of the major effects of System/360 was to establish architecture as an autonomous function and to introduce the management tools, discipline, and procedures for adopting and controlling architecture [9].

Recognizing that any differences in wording may imply differences in function, consistency is achieved by having only one specification of the architecture; it tells IBM machine designers the functions the machine must provide, and it describes to IBM programmers how the machine operates. The same specification is made available outside IBM as the *Principles of Operation* [16, 18], and is the only authoritative specification that describes the architecture. An analogous specification exists for the I/O interface [24].

A set of procedures have been established for the development of an architecture, starting with the conception of the idea and ending with the formal adoption of a definition. These procedures provide for the assessment of the cost and value of a function and for the approval of the architecture by machine and software implementers. Rules have been established about the extent of architectural compatibility [16], and provision is made for deviating from the common definition.

Although the implementation of a line of compatible computers did not take an undue amount of effort, the design and control of architecture proved to require more attention to detail than originally anticipated. Furthermore, experience with System/360 and its subsequent extensions has shown that the management of architecture must be an ongoing operation to ensure a consistent technical interpretation and to ensure that the evolution of the architecture structure is governed by a consistent set of principles and a design philosophy.

Conclusion

System/360 architecture has provided the basis for a number of machine generations, and it has been able to evolve to respond to new technologies, programming-system structures, and user requirements. This has been possible because of the soundness of its basic structure, the rigorousness of its definition, and the recognition of the autonomy of the architecture function.

As machine, software, and system-design technologies advance, further evolution of the architecture is inevitable. Changes will be made to better meet user needs and to allow more efficient design of machines and their associated programming systems. Because of the magnitude of the investment in System/370 architecture, however, it will be even more essential to ensure compatibility with the current architecture for those interfaces that are exposed to the user and are intended for application-program development.

Appendix: Model characteristics

This appendix summarizes some attributes of IBM machines implementing System/360 and System/370 architecture. Only the most recent characteristics are listed; some of the models were improved after initial announcement. The tables in this section are updated and extended versions of those published by Case and Padegs [9], and some corrections have been included.

Table 1 (appearing on pages 388–389) lists some key characteristics of the CPU and storage. CPU data-flow width indicates the largest field that can be handled in one cycle time. Depending on the CPU, a different amount of "work" is accomplished per CPU cycle; hence the cycle time cannot be used directly as a measure of relative speed.

Control storage contains the microprogram. A range in the size is given for those models where the amount installed depends on the selection of certain optional features. The word size is expressed in terms of two numbers: (the number of bits used for logic or control purposes) + (the number of bits used for checking the parity of the control-storage contents). When a separate control storage is provided for the service processor or channels, the table lists only the parameters of the CPU control storage.

The bus width for some models is expressed in terms of two numbers: (basic width) × (interleaving factor). The basic width is the width of the path from the storage controller to the CPU or channels. The interleaving factor indicates the number of accesses to sequential locations that can be made in one storage cycle; it applies to implementations where sequential locations are in different storage modules. The interleaving factor may be variable and may depend on the configuration. On some models the bus width is smaller than the amount of information accessed in parallel in the storage array; this is indicated by footnotes. Unless otherwise indicated, the storage-cycle time is the minimum time between successive references to the same location.

For the cache, the line-width column gives the number of bytes in the cache which are considered as one unit for addressing and replacement purposes. The first element of the product notation is the minimum transfer unit from processor storage to cache; the second element is the number of such transfer units required to make a line. Where applicable, a two-number notation is used for the cycle time to indicate the minimum time between successive read accesses and the total cache-access time. Usually, the contents of a particular virtual address in storage may be placed in only a small part of the available cache locations, where they may be found by an associative lookup. The column labeled "Associativity" shows the degree of associativity, that is, the number of different locations in the cache that may correspond to a particular virtual address.

Table 2 lists the year, month, and day when the various machines were announced and the year and month when they were first shipped.

References and notes

1. The term *architecture* is used here to describe the attributes of a system as seen by the programmer, *i.e.*, the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation.
2. G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM J. Res. Develop.* **8**, 87–101 (1964).
3. G. A. Blaauw and F. P. Brooks, Jr., "The Structure of System/360; Part I—Outline of the Logical Structure," *IBM Syst. J.* **3**, 119–135 (1964).
4. G. M. Amdahl, "The Structure of System/360; Part III—Processing Unit Design Considerations," *IBM Syst. J.* **3**, 144–164 (1964).
5. A. Padegs, "The Structure of System/360; Part IV—Channel Design Considerations," *IBM Syst. J.* **3**, 165–180 (1964).
6. G. A. Blaauw, "The Structure of System/360; Part V—Multisystem Organization," *IBM Syst. J.* **3**, 181–195 (1964).
7. A. Padegs, "Structural Aspects of the System/360 Model 85; Part III—Extension to Floating-point Architecture," *IBM Syst. J.* **7**, 22–29 (1968).
8. D. T. Brown, R. L. Eibsen, and C. A. Thorn, "Channel and Direct Access Device Architecture," *IBM Syst. J.* **11**, 186–199 (1972).
9. R. P. Case and A. Padegs, "Architecture of the IBM System/370," *Commun. ACM* **21**, 73–96 (1978).
10. C. J. Bashe, W. Buchholz, G. V. Hawkins, J. J. Ingram, and N. Rochester, "The Architecture of IBM's Early Computers," *IBM J. Res. Develop.* **25**, 363–375 (1981, this issue).
11. W. Buchholz, Ed., *Planning a Computer System (Project Stretch)*, McGraw-Hill Book Co., Inc., New York (1962).
12. D. W. Sweeney, "An Analysis of Floating-Point Addition," *IBM Syst. J.* **4**, 31–42 (1965).
13. W. C. Carter, H. C. Montgomery, R. J. Preiss, and H. J. Reinheimer, "Design of Serviceability Features for the IBM System/360," *IBM J. Res. Develop.* **8**, 115–126 (1964).
14. C. T. Gibson, "Time-Sharing in the IBM System/360: Model 67," *AFIPS Conference Proceedings* **28** (1966 Sprint Joint Computer Conference, Boston), 61–78 (1966).
15. G. R. Blakeney, L. F. Cudney, and C. R. Eickhorn, "An Application-Oriented Multiprocessing System, Part II—Design Characteristics of the 9020 System," *IBM Syst. J.* **6**, 80–94 (1967).
16. *IBM System/370 Principles of Operation*, Order No. GA22-7000, available through IBM branch offices.
17. M. Y. Hsiao, W. C. Carter, J. W. Thomas, and W. R. Stringfellow, "Reliability, Availability, and Serviceability of IBM Computer Systems: A Quarter Century of Progress," *IBM J. Res. Develop.* **25**, 453–465 (1981, this issue).
18. *IBM 4300 Processors Principles of Operation for ECPS:VSE Mode*, Order No. GA22-7070, available through IBM branch offices.
19. H. R. Schwermer, "The ECPS:VSE Mode for the IBM 4300 Processors," *IEEE COMPCON Spring 1980 Digest of Papers*, San Francisco, Feb. 25–28, 1980.
20. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, "One-Level Storage System," *IRE Trans. Electron. Computers* **11**, 223–235 (1962).
21. J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems," *J. ACM* **12**, 589–602 (1965).
22. B. W. Arden, B. A. Galler, T. C. O'Brien, and F. H. Westervelt, "Program and Addressing Structure in a Time-Sharing Environment," *J. ACM* **13**, 1–16 (1966).
23. Only instructions published in the *Principles of Operation* are included in the counts. The following are not included: instructions available on a special-contract basis, instructions that are part of assists for specific operating systems, and instructions for emulating other architectures. For System/360, the special instructions available only on Models 20, 44, and 67 are not included. For System/370, instructions associated with the one-level-addressing facility are not included.
24. *IBM System/360 and System/370 I/O Interface: Channel to Control Unit, Original Equipment Manufacturer's Information*, Order No. GA22-6974, available through IBM branch offices.
25. P. Fagg, J. L. Brown, J. A. Hipp, D. T. Doody, J. W. Fairclough, and J. Greene, "IBM System/360 Engineering," *AFIPS Conference Proceedings* **26** (1964 Fall Joint Computer Conference, San Francisco), 205–231 (1964).
26. W. Y. Stevens, "The Structure of System/360; Part II—System Implementations," *IBM Syst. J.* **3**, 136–143 (1964).
27. M. V. Wilkes, "The Best Way to Design an Automatic Calculating Machine," *Manchester University Computer Inaugural Conference*, Manchester, England, 1951, p. 16.
28. S. G. Tucker, "Microprogram Control for System/360," *IBM Syst. J.* **6**, 222–241 (1967).
29. S. S. Husson, *Microprogramming Principles and Practice*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1970.
30. P. M. Davies, "Readings in Microprogramming," *IBM Syst. J.* **11**, 16–40 (1972).
31. S. G. Tucker, "Emulation of Large Systems," *Commun. ACM* **8**, 753–761 (1965).

Table 2 Announcement and shipment dates.

Model	*Announced*	*First shipped*
System/360		
22	71-4-7	71-6
25	68-1-3	68-10
30	64-4-7	65-6
40	64-4-7	65-4
44	65-8-16	66-9
50	64-4-7	65-8
60	64-4-7	not shipped[1]
62	64-4-7	not shipped[1]
65	65-4-22	65-11
67	65-8-16	66-5
70	64-4-7	not shipped[2]
75	65-4-22	66-1
85	68-1-30	69-12
91	64-11-17	67-10
92	64-8-17	not shipped[3]
95	*	68-2
195	69-8-20	71-3
System/370		
115	73-3-13	74-3
115-2	75-11-10	76-4
125	72-10-4	73-4
125-2	75-11-10	76-2
135	71-3-8	72-4
135-3	76-6-30	77-2
138	76-6-30	76-11
145	70-9-23	71-6
145-3	76-6-30	77-5
148	76-6-30	77-1
155	70-6-30	71-1
158	72-8-2	73-4
158-3	75-3-25	76-9
165	70-6-30	71-4
168	72-8-2	73-5
168-3	75-3-25	76-6
195	71-6-24	73-8
System/370-compatible		
3031	77-10-6	78-3
3032	77-10-6	78-3
3033	77-3-25	78-3
3033-N	79-11-1	80-1
3033-S	80-11-12	
3081	80-11-12	
4331-1	79-1-30	79-3
4331-2	80-5-7	80-8
4341-1	79-1-30	79-11
4341-2	80-9-15	

[1]Replaced by Model 65
[2]Replaced by Model 75
[3]Redesignated as Model 91
*Offered on special government contract

Table 1 Model characteristics.

Model	CPU		Control storage				Number of TLB entries	Processor storage			Cache				
	Data-flow width (bytes)	Cycle time (ns)	Size (K words)	Word size (bits)	Type	Cycle time (ns)		Size (K bytes)	Bus width (bytes)	Cycle time (ns)	Size (K bytes)	Line width (bytes)	Cycle time (ns)	Type	Associativity
System/360															
22	1	750	4	50+5	RO	750	none	24-32	1	1500[1]	none				
25	1	900	8	16+2	RW	900	none	16-48	2	900[1]	none				
30	1	750	4	50+5	RO	750	none	16-64	1	1500[1]	none				
40	2[2]	625	4	52+2	RO	625	none	32-256	2	2500[1]	none				
44	4	250	none				none	32-256	4	1000[1]	none				
50	4	500	2.75	85+3[3]	RO	500	none	128-512	4	2000[1]	none				
								1024-8192	4×(1-2)	8000[1]					
65	8	200	2.75	87+4[4]	RO	200	none	256-1024	8×2	750[1]	none				
								1024-8192	8×(1-2)	8000[1]					
67	8	200	2.75	87+4[4]	RO	200	8	256-1024	8×2	750[1]	none				
75	8	195	none				none	256-1024	8×(2-4)	750[1]	none				
								1024-8192	8×(1-2)	8000[1]					
85	8	80	2	105+3[5]	RO	80	none	512-4096	16×(2-4)	960[1]	16-32	16×4[6]	80-160	T	16
			0.5	105+3[5]	RW	80									
91	8	60	none				none	2048-6144	8×16	780[1]	none				
95	8	60	none				none	1024	8×16	180	none				
								1024-6144	8×16	780[1]					
195	8	54	none				none	1024-4096	8×(8-16)	756[1]	32	8×8	54-162	T	4
System/370															
115	1	480	20-28	20+3	RW	480	8	64-192	2	480	none				
115-2	2	480	12-20[7]	19+2	RW	480	16	64-384	2	480	none				
125	2	480	12-20	19+2	RW	480	16	96-256	2	480	none				
125-2	2	320-480[8]	16-24	19+2	RW	320	16	96-512	2	480	none				
135	2	275-1485[8]	12-24	16+2	RW	275	8	96-512	2	990[9] R 935[9] W	none				
135-3	2	275-1485[8]	64	16+2	RW	275	8	256-512	2	990[9] R 935[9] W	none				
138	2	275-1430[8]	64	16+2	RW	275	8	512-1024	2	935[9]	none				
145	4[10]	203-315[8]	8-16[11]	32+4	RW	203	8	160-2048	8	540 R 608 W	none				
145-3	4[10]	180-270[8]	32	32+4	RW	180	8	192-1984	8	405 R 540 W	none				
148	4[10]	180-270[8]	32	32+4	RW	180	8	1024-2048	8	405 R 540 W	none				
155	4	115	6	69+3	RO	115	none	256-2048	8	2070[1]	8	16	115-230	T	2
155-II	4	115	8	69+3	RO	115	128	256-2048	8	2070[1]	8	16	115-230	T	2
158	4	115	8	69+3	RW	115	128	512-6144	8	1035 R 920 W	8	16	115-230	T	2
158-3	4	115	8	69+3	RW	115	128	512-6144	8	920	16	16×2	115-230	T	4
165	8	80	2	105+3	RO	80	none	512-3072	8×4	2000[1]	8-16	8×4	80-160	T	4
			2	105+3[5]	RW	80									
165-II	8	80	4	105+3[5]	RO	80	128	512-3072	8×4	2000[1]	8-16	8×4	80-160	T	4
			1	105+3[5]	RW	80									
168	8	80	2-3.5	105+3[5]	RO	80	128	1024-8192	8×4	320	8-16	8×4	80-160	T	4-8[12]
			0.5-1	105+3[5]	RW	80									
168-3	8	80	2-3.5	105+3[5]	RO	80	128	1024-8192	8×4	320	32	8×4	80-160	T	8
			1-2	105+3[5]	RW	80									
195[13]	8	54	none				none	1024-4096	8×16	756	32	8×8	54-162	T	4
System/370-compatible															
3031	4	115	8	69+3	RW	115	128	2048-8192	8×4	345[14]	32	8×4	115-230	T	8
3032	8	80	4	105+3	RW	80	128	2048-8192	8×4	320	32	8×4	80-160	T	8
3033	8	57	3-7	105+3	RW	57	128	4096-25576	8×8	285[14]	64	8×8	57-114	T	16
			1	122+4	RW	57									
3033-N	8	57	3-7	105+3	RW	57	128	4096-16384	8×4	285	16	8×8	57-114	T	8
			1	122+4	RW	57									
3033-S	8	57	3-7	105+3	RW	57	128	4096-8192	8×4	285	0.5	8×4	57-114	T	8
			1	122+4	RW	57									
3081[15]	8	26	2[16]	104+4	RW	52[17]	128	16384-32768	8×2[18]	312[19]	32	8×16	26-52	C	4
4331-1	4	300-1600[8]	16-32[11]	32+4	RW	500[20]	64	512-1024	4	900 R 1300 W	none				
4331-2	4	200-1600[8]	32[11]	32+4	RW	500[20]	64	1024-4096	4	2600[21] R 3100[21] W	8	4×16	200	C	4
			3	32+4	RO	100									
4341-1	8	150-300[8]	14-16	32+4	RW	150	64	2048-4096	8	2400[21]	8	8×8	225	C	4
4341-2	8	120-240[8]	16-20	32+4	RW	120	64	2048-8192	16	1440[21]	16	16×4	120 R 180 W	C	8

Explanation

- **C** Store-in-cache: On storing, the value is placed in the cache; the new value is placed in main storage at the time the cache line is reassigned or the data is requested by a channel or another processor.
- **K** The number $2^{10} = 1024$
- **ns** Nanoseconds
- **R** Access for reading
- **RO** Read-only
- **RW** Read-write (writable)
- **T** Store-through: On storing, the value is placed in main storage; the value is not placed in the cache unless a line has been assigned to the main-storage location.
- **TLB** Translation-lookaside buffer, which is a part of the dynamic-address-translation mechanism
- **W** Access for writing

Footnotes

[1] The model uses magnetic-core technology.

[2] Certain registers and paths are 17 or 18 bits wide where a main-storage address is processed in one cycle.

[3] Extended to 90 + 3 for the 1410 emulator, or 92 + 3 for the 7070 emulator.

[4] Extended to 94 + 4 when any emulator is installed.

[5] Extended to 122 + 4 for part of control storage when any emulator is installed.

[6] Although the 64-byte lines are loaded into the cache only when referred to, an entire cache sector of 1K bytes (16 lines) is assigned as a unit to a 1K-byte storage sector.

[7] The 115-2 contains a separate I/O processing unit for some functions that were executed on the CPU in a 115; hence the smaller CPU control-storage capacity.

[8] Variable, depending on the type of operation performed.

[9] Four bytes can be accessed and transferred in this time.

[10] An 8-byte-wide path is used for instruction fetch.

[11] Part of this capacity is physically in processor storage and thus has to be subtracted from the available processor-storage capacity.

[12] Depends on cache size used.

[13] The System/370 Model 195 has certain facilities (*e.g.*, time-of-day clock, control registers, MOVE LONG) not available on the System/360 Model 195.

[14] The effective transfer rate to the CPU is limited to eight bytes per CPU cycle.

[15] Each of the two CPUs has the indicated control-storage, cache, and TLB capacity.

[16] 1K of the control storage is pageable, using an area in processor storage assigned for this purpose.

[17] 26 ns when the word is available in the microinstruction buffer (*i.e.*, is within the current set of 16 words).

[18] Interleaving is on the basis of 2K bytes; no interleaving takes place within the access for a cache line of 128 bytes.

[19] An amount equal to a cache line is read or written in one storage cycle. The effective transfer rate to the CPU is limited to eight bytes per CPU cycle.

[20] 100 ns when the word is available in the microinstruction buffer.

[21] An entire cache line can be accessed and transferred between the cache and the storage unit in this time.

Jean E. Sammet

History of IBM's Technical Contributions to High Level Programming Languages

This paper discusses IBM's technical contributions to high level programming languages from the viewpoint of specific languages and their contributions to the technology. The philosophy used in this paper is that it is the appropriate collection of features in a language which generally makes the contribution to the technology, rather than an individual feature. Those IBM languages deemed to have made major contributions are (in alphabetical order) APL, FORTRAN, GPSS, and PL/I. Smaller contributions (because of lesser general usage) have been made by Commercial Translator, CPS, FORMAC, QUIKTRAN, and SCRATCHPAD. Major contributions were made in the area of formal definition of languages, through the introduction of BNF (Backus-Naur Form) for defining language syntax and VDL (Vienna Definition Language) for semantics.

1. Introduction

This paper delineates some of IBM's technical contributions to high level programming languages, with some discussion of related work outside IBM to provide perspective. Section 2 provides a brief but broad chronological tracing of high level language developments, and Section 3 discusses the two very early IBM languages. Of all the other languages developed by IBM, four are major because they were widely used, as well as making significant technical contributions; they are described in Section 4. The four (listed alphabetically) are APL, FORTRAN, GPSS, and PL/I. Other languages (*e.g.*, FORMAC, Commercial Translator) had significant impact on the technology and/or on the development of other languages but never became major in their own right; those are discussed in Section 5. Still others made lesser conceptual contributions to the field (*e.g.*, COURSEWRITER) and are briefly mentioned in Section 7. Section 6 deals with formal definition methodology.

Aside from Section 2, which provides a very broad framework for the whole field of high level programming languages, this paper does not discuss the IBM languages in strict chronological order. The reason for this is that there was relatively little interdependence among the IBM languages. While each language used whatever was appropriate from existing or prior language technology both inside and outside IBM, there was no direct upward technical progression, as occurs in some other aspects of the computer field. Furthermore, the thrust of this paper is on each language as a unit, rather than its component elements. In most cases, it was really the proper technical packaging of ideas—some old, some new—in each language which made the contribution. Stated more explicitly, it is my view that the significant technical contributions made in the programming language area are by the cohesive combination of features in a language, and not particularly by an individual feature in a single language regardless of its novelty.

In addition to specific languages, there have been technical contributions from IBM in related fields. The whole subject of compilers is being covered in another paper [1] in this issue. However, methodologies for language definition are mentioned in Section 6 of this paper; the Vienna Definition Language is discussed in more detail in [2] in this issue.

The definition of the term "programming language" has been unclear and controversial almost from the beginning of computer activity. In this paper, "programming language" is used as the equivalent of "high level language." The latter was defined in [3] to have the following four characteristics:

1. Knowledge of machine code is unnecessary.
2. There is good potential for converting a program written in high level language for one computer to run on another computer with minimal difficulty.
3. There is an instruction expansion, *i.e.*, a single statement in a high level language will produce many machine code instructions.
4. The notation for the language is problem oriented, *i.e.*, it is closer to the original conceptual statement of the problem than are machine instructions.

The last point in particular is meant to exclude from the high level language category any system involving fixed fields and fixed formats. This eliminates RPG and decision tables from the category of high level languages. It must be emphasized that this is a technical taxonomy of high level languages and not a value judgment of the usefulness of RPG or decision tables or any other tool or technique with which the user communicates with the computer. Also excluded are assembly languages (even with macros), languages for doing microprogramming, command languages, and text editors. Again, there is no value judgment intended here of the importance or value of these facilities but merely an attempt to keep within the definition. Finally, a significant set of subroutines added to an existing language, but without any change to the base language (*e.g.*, SLIP [4]), is not considered a new language.

The general area of application programming tools, some of which border on high level languages, is also outside the scope of this paper.

Because many of the source documents referred to in this paper are not publicly available, and because most of the languages mentioned in this paper have been described in the author's book [3], many citations to the book are used to provide an easily accessible source for the reader to find both a description of the language and the primary references.

Naturally, the comments in this paper and the value judgments expressed or implied are the personal views of the author and do not represent an official view of the IBM Corporation.

2. Broad chronology

In order to provide perspective on IBM's technical contributions to programming languages, we trace briefly their overall development. (A more detailed history covering the period through 1971 is given in [5].) Later sections of this paper discuss each of the specific IBM languages.

The earliest work known which legitimately fits the programming language definition given earlier is the "Plankalkül" by Zuse in Germany (1945). Unfortunately, this was not implemented. The next step was Short Code, suggested by J. Mauchly and implemented by others at Remington Rand UNIVAC (1949–50), and then the unnamed and unimplemented language developed by Rutishauser in Switzerland (1952). The Speedcoding system for the IBM 701 was developed by IBM in 1953 and is discussed in Section 3. In this time frame, Remington Rand produced the A-2 and A-3 systems (based on three-address pseudocodes to indicate mathematical operations), and the Boeing Company developed BACAIC for the 701. All these languages, plus others of that period, attempted to provide scientists and engineers with a notation slightly more natural to mathematics than machine code. Some permitted the users to write mathematical expressions in relatively normal format, but most did not. None of them had any significant or lasting effect on language development, and apparently minimal effect (if any) on people's thinking.

In May 1953 J. H. Laning, Jr., and N. Zierler at MIT had a system running on Whirlwind that appears to be the first system in the United States to permit the user to write expressions in a notation resembling normal mathematical format, *e.g.*,

$$c = 0.0052\ (a - y)/2ay,$$

$$y = 5y,$$

An excellent description of most of these early mathematical systems, plus others, appears in [6], and some of them are described briefly in [3], and even more briefly in [7].

In 1954, work on FORTRAN started, and the 704 compiler was released in April 1957. PRINT (described in Section 3) was actually finished before FORTRAN.

About the time the preliminary FORTRAN report was issued, a group at Remington Rand UNIVAC under Grace Hopper's direction began development of a system originally called AT-3 and later renamed MATH-MATIC. (John Backus says that he sent them a copy of his November 1954 preliminary FORTRAN report, but I cannot determine how much influence it had on MATH-MATIC. The preliminary FORTRAN specifications precede any language design documents on MATH-MATIC from Remington Rand UNIVAC.) MATH-MATIC (described in [3, 6]) was similar in spirit to FORTRAN, although different in syntax, and it is

not clear which system was actually running first. However, of all the parallel work going on in the mid-1950s, only FORTRAN has survived, and by 1957 there were the first glimmerings of significant practical usage of a high level language similar to those we know today.

While the main emphasis prior to 1958 was on the development of languages for scientific applications, the first English-like language for business data processing problems, FLOW-MATIC, was planned and implemented on the UNIVAC I under the direction of Grace Hopper at Remington Rand UNIVAC and released in 1958. (A description appears in [3].)

Activity on another language, APT, started in 1956 at MIT under Douglas T. Ross. APT was for numerical machine tool control, and hence was the first language for a specialized application area. (See [8] for full details on the early development.) APT (albeit modified over time) was still in use in 1980.

The years 1958 and 1959 were among the most prolific for the development of programming languages. The following events in universities and industrial organizations all occurred during those two years:

1. The development of the IAL (International Algebraic Language), which became known as ALGOL 58, and the publication of its definition [9]. IAL had a profound effect on the computing world, because of
   a. Its follow-on, ALGOL 60, which is clearly one of the most important languages ever developed, and
   b. The development of three languages based on the IAL specifications, namely, NELIAC, MAD, and CLIP (which eventually was the foundation for JOVIAL). All except CLIP became widely used.
2. The availability in early 1958 of a running version of IPL-V (a list processing language).
3. The start of work on the development of LISP, a list processing language which was intended for artificial intelligence applications.
4. The first implementation of COMIT, a string processing language.
5. The formation in May 1959 of the CODASYL (Conference On Data SYstems Languages) Short Range Committee, which developed COBOL, and the completion of the COBOL specifications.
6. The development and availability of language specifications for AIMACO, Commercial Translator (see Section 5), and FACT, all of which were for business data processing problems.
7. The availability of specifications for JOVIAL.

Of all these languages from 1958–59 and earlier, those that survive (albeit in modified form) and have significant usage in 1980 are ALGOL 60, APT, COBOL, FORTRAN, JOVIAL, and LISP. References and a discussion of all the languages named in this section can be found in [3]. A detailed history of the development of the six languages just cited is provided in [10].

A large impetus for most of this work was economic—even then programming costs were large, and any steps or tools which could reduce those costs were looked at favorably. However, the crucial issue often was whether any "slowdown" caused by these systems exceeded the overall savings in people's money or time; generally the answers favored the use of such systems.

The period 1960–1970 saw some maturation of the programming language field. The material for this period is taken almost verbatim from [5], copyright 1972, Association for Computing Machinery, by permission; descriptions of and references for the languages mentioned are in [3]. During this time the battle over the use of high level languages was clearly won, in the sense that machine coding had become the exception rather than the rule. (This comment is based only on the author's opinion and perception because there is simply no data to verify or contradict this statement.) Although the concept of developing systems programs by using high level languages was fairly well accepted, there was more machine coding of systems programs than of application programs. The use of powerful macro systems and "half way" languages such as PL/360 [11] provided some of the advantages of high level languages but made no attempt to be machine-independent.

The major new batch languages of this decade were ALGOL 60, COBOL, and PL/I, of which only the last two were extensively used in the United States. Although ALGOL 68 was defined, its implementation was just starting around 1970.

The advent of interactive programming in the mid-60s brought a host of on-line languages, starting with JOSS and later followed by BASIC, both of which became very widely used. Each had many imitators and extenders. APL\360 (see Section 4) was made available late in the 1960s and became popular among certain specific groups.

The development of high level languages for use in formula manipulation was triggered by FORMAC (see Section 5) and Formula ALGOL, although only the former was widely used. String processing and pattern matching became popular with the advent of SNOBOL.

The simulation languages GPSS (see Section 4) and SIMSCRIPT made computer simulation more accessible to most users and also encouraged the development of other simulation languages. A number of other languages for specialized application areas (*e.g.*, civil engineering, equipment checkout) continued to be developed.

Perhaps one of the most important practical developments in this time period, although scorned by many theoreticians, was the development of official standards for FORTRAN and COBOL and the start of standardization for PL/I.

The period 1970–1980 involved relatively few significant pure language developments. Those few include: (1) the implementation and initial limited usage of ALGOL 68; (2) the implementation and heavy use of Pascal; (3) the massive effort by the Department of Defense to develop a single language—called Ada—for embedded computer systems [12, 13]; (4) the concept of data abstraction; and (5) concepts of functional programming by Backus [14], and (6) experimental languages such as CLU, EUCLID, and SETL. It is too early to tell which—if any—of these concepts and languages will have a fundamental effect on the computer field.

3. Very early IBM languages

The earliest IBM attempt at what was then considered a high level language was Speedcoding for the IBM 701 [15]. The motivation for its development was similar to that for all the early languages—to provide the user with a notation which was easier to use than raw machine code. Work was started on Speedcoding in January 1953 under the supervision of John Backus and the general direction of John Sheldon; the first official manual was dated September 1953.

The basic principle of Speedcoding was to create two sets of operations, the first category containing three addresses and the second set containing only one. The operations were not part of the hardware and were selected for their utility to the mathematician. Thus

```
523 SUBAB 100 200 300 TRPL 500
```

shows an instruction in location 523 which causes the computer to subtract the absolute value of the contents of location 200 from the value in 100 and to put the result in 300; then the computer tests the sign of the result in 300 and transfers control to 500 if it is positive.

Such a notation looks primitive by standards in use only a few years after its development, but it was actually used on many 701s and may have influenced the designers of the 704. However, Speedcoding had no lasting language effect.

The other early language of IBM was PRINT (PRe-edited INTerpretive System) [3]. It was designed under the leadership of Robert Bemer and was intended to meet the scientific computing needs of IBM 705 users. Since the 705 had been designed primarily for use in business data processing, PRINT was created to meet the needs of people who wished to use it for scientific computing. It provided a series of operation codes with variable fields (which could contain one to four variables, depending on the operation code).

Coding of the PRINT system started in February 1956, and the first customer tried it in July 1956. Thus, it was actually completed before FORTRAN. PRINT seems to have had no significant technical effect on other languages.

4. Major IBM languages

The question of which languages (of the many hundreds developed since 1950) might be construed as being "major" is highly controversial. The judgment used in this paper reflects that of a very well qualified group of individuals who served as the Program Committee for the ACM SIGPLAN History of Programming Languages Conference held in June 1978. To provide perspective for the categorizations made in this paper, we quote [10, p. *xviii*] the rationale used (in 1977) for selecting the languages for that conference and list the languages which were accepted. "The specific requirements were that the languages (1) were created and in use by 1967; (2) remain in use in 1977; and (3) have had considerable influence on the field of computing. (The cut-off date of 1967 was chosen to provide perspective from a distance of at least ten years.) The general criteria for choosing the languages are the following (not necessarily in order of importance, nor required to have each one apply to each language): Usage, influence on language design, overall impact on the environment, novelty (first of its kind), and uniqueness."

The program committee selected the following languages as satisfying the indicated criteria: ALGOL, APL, APT, BASIC, COBOL, FORTRAN, GPSS, JOSS, JOVIAL, LISP, PL/I, SIMULA, SNOBOL.

For two of these languages, FORTRAN and GPSS, the first version (and some of the later ones) were designed and implemented solely within IBM. A third, PL/I, involved a joint effort of IBM and SHARE for the language development, but the first commercial implementation was done entirely by IBM. APL was designed initially by K. Iverson at Harvard University, but its design was continued and modified after he joined IBM and was then implemented by IBM. And, of course, IBM employees contributed significantly to the external committee developments of ALGOL 58, ALGOL 60, and COBOL. The portion of this paper describing the four "major IBM languages" is based primarily on the papers prepared by the authors for the conference. (See [10, 16].)

Figure 1 Table of FORTRAN I statements for the IBM 704. The spacing is not significant.

Statement	*Normal sequencing*
$a = b$	Next executable statement
GO TO n	Statement n
GO TO n, (n_1, n_2, . . . , n_m)	Statement last assigned
ASSIGN i TO n	Next executable statement
GO TO (n_1, n_2, . . . , n_m), i	Statement n_i
IF (a) n_1, n_2, n_3	Statement n_1, n_2, n_3 as a less than, =, or greater than 0
SENSE LIGHT i	Next executable statement
IF (SENSE LIGHT i) n_1, n_2	Statement n_1, n_2 as Sense Light i ON or OFF
IF (SENSE SWITCH i) n_1, n_2	Statement n_1, n_2 as Sense Switch i DOWN or UP
IF ACCUMULATOR OVERFLOW n_1, n_2	Statement n_1, n_2 as Accumulator Overflow trigger ON or OFF
IF QUOTIENT OVERFLOW n_1, n_2	Statement n_1, n_2 as MQ Overflow trigger ON or OFF
IF DIVIDE CHECK n_1, n_2	Statement n_1, n_2 as Divide Check trigger ON or OFF
PAUSE or PAUSE n	Next executable statement
STOP or STOP n	Terminates program
DO n $i = m_1$, m_2 or DO n $i = m_1$, m_2, m_3	Next executable statement
CONTINUE	" " "
FORMAT (*Specification*)	Not executed
READ n, *list*	Next executable statement
READ INPUT TAPE i, n, *list*	" " "
PUNCH n, *list*	" " "
PRINT n, *list*	" " "
WRITE OUTPUT TAPE i, n, *list*	" " "
READ TAPE i, *list*	" " "
READ DRUM, i, j, *list*	" " "
WRITE TAPE i, *list*	" " "
WRITE DRUM i, j, *list*	" " "
END FILE i	" " "
REWIND i	" " "
BACKSPACE i	" " "
DIMENSION v, v, v, . . .	Not executed
EQUIVALENCE (a,b,c, . . .), (d,e,f, . . .),	" "
FREQUENCY n (i,j, . . .), m(k,l, . . .), . . .	" "

• *FORTRAN*

As mentioned in Section 2, there were a number of early attempts at "automatic programming" systems (which is the term used in the 1950s for systems of this type). However, John Backus in [7, p. 26] states that "The Laning and Zierler system was . . . the world's first operating algebraic compiler, a rather elegant but simple one." He also describes the sequence of events which finally made it clear to him that the Laning and Zierler work did not influence FORTRAN. (Earlier, Backus had publicly stated his belief that the algebraic nature of FORTRAN was derived from the Laning and Zierler work.)

Backus indicates that in those days programmers were proud of their ability to use the computer efficiently. Many of them believed that any attempt to automate programming would lead to intolerable inefficiencies. It is important to understand that milieu because it significantly influenced the design philosophy for FORTRAN. Backus points out in [7, p. 27] that even at that time economic considerations made it clear that steps taken to reduce programming costs would be of value. He said that: "This economic factor was one of the prime motivations which led me to propose the FORTRAN project in a letter to my boss, Cuthbert Hurd, in late 1953."

Backus specifically identified [7, p. 28] the systems that existed at that time and their influence (almost none) on the development of FORTRAN. He then indicated that the 704 presented challenges to the designers of any new software system which attempted to simplify programming, because the 704 included floating point arithmetic and index registers whose simulation was part of the justification for some of the earlier software systems. He then said [7, p. 28]: "In view of the widespread skepticism about the possibility of producing efficient programs with an automatic programming system and the fact that inefficiencies could no longer be hidden, we were convinced that the kind of system we had in mind would be widely used only if we could demonstrate that it would produce programs almost as efficient as hand coded ones and do so on virtually every job." Consequently the design of the language as such was relatively straightforward, and the major emphasis was on a system which would produce efficient object code.

In order to provide an appropriate time sequence, note that a preliminary FORTRAN report was issued in November 1954, a reference manual was issued in October 1956, and a primer in early 1957. External professional publication occurred in 1957 in [17]. It is interesting to contrast the FORTRAN I statements with later and current versions of the language. Figure 1 shows all the FORTRAN I statements.

One of the interesting myths about FORTRAN that Backus puts to rest in his paper is the relation between the FORTRAN subscripts and the index registers on the 704. It had been widely thought for many years that only three subscripts were allowed in FORTRAN because there were only three index registers on the 704. Backus makes it clear [7, p. 34] that the reason for the limitation to three was the complexity of attempting to optimize index register allocation.

One of the most interesting (and novel) statements in FORTRAN I was FREQUENCY, which allowed the programmer to specify which path on a branch he thought was most likely. The compiler then attempted to optimize object code based on that information. It apparently worked well, but people usually did not have the correct data about the best branch and often used it in circumstances that made little difference; it was removed in FORTRAN IV. It should also be noted that FORTRAN I (and even FORTRAN II, discussed below) contained some machine-oriented statements, namely, references to tape/drum, sense switches, and even the accumulator (for overflow). Eventually these were dropped, when machine independence became more significant than it was in the early versions. The EQUIVALENCE statement in FORTRAN I provided a very early attempt to enable the programmer to use storage efficiently. The use of the loop statement DO became a model for all future languages (although most later languages significantly enhanced the FORTRAN loop statement capability).

FORTRAN II introduced two important concepts into the language—subroutines and the COMMON statement to permit communication among subroutines; it was also possible to link to assembly language programs. (Although subroutines had been in use for many years, their introduction into a high level language was an important step.)

Although neither FORTRAN I nor II had separate data declarations as such, FORTRAN I did use the concept indirectly, by specifying that variables whose names began with the letters I, J, K, L, M, or N were considered integers and all others were considered floating point.

A lot of cleanup and elimination of machine-dependent features was done in FORTRAN IV, and, along with FORTRAN II as a subset, it formed the basis for the first ANSI language standard(s) [18, 19].

In my own view, FORTRAN has probably had more impact on the computer field than any other single software development. Its major technical contribution was to demonstrate that efficient object code could be produced by a compiler; as a result, it became clear that productivity of programmers could be significantly improved. The fact that FORTRAN still exists in spite of more modern languages with newer concepts is testimony to the soundness of many of the original ideas (but also to inertia and investment in old programs). Naturally, since the first version of FORTRAN, a number of additions and changes have been made and presumably will continue to be made. But the basic framework was sound and enabled many people to build upon it.

This is a reasonable place to indicate some of the relationships between ALGOL and FORTRAN, particularly since Backus (of IBM) was one of the four U.S. representatives to the IAL (= ALGOL 58) activity; he also participated in the ALGOL 60 development. Since both FORTRAN I and II were released before the ALGOL 58 committee finished its language design [9], ALGOL 58 could not have any effect on FORTRAN. Space does not permit a discussion here of the effect FORTRAN had on ALGOL 58. In my view, neither of the ALGOL versions had any significant effect on FORTRAN IV.

• *GPSS (General Purpose Simulation System)*

GPSS was developed by Geoffrey Gordon. He had worked on simulation studies in various places prior to joining IBM, including Bell Telephone Laboratories, where he participated in the development of a program called the Sequence Diagram Simulator, which was presented at a meeting of the IEEE in 1960. When he joined IBM in June 1960, it was in the Mathematics and Programming Department under D. V. Newton.

The work being done in that department involved queuing models. Gordon suggested developing a system description language based on the Sequence Diagram Simulator approach. He began writing a program implementing a block diagram language and chose a relatively simple table-oriented interpretive structure that would make changes easier. It was written using the SAP assembly language for the IBM 704. Gordon stated [20, p. 407]

that he was not aware of the work on SIMSCRIPT being done about the same time.

GPSS is a discrete simulation language, and the major characteristic that has made it useful twenty years after its initial development is that the block diagram portion of the language provides an excellent means of communication with systems people.

The most relevant similar development is that of SIMSCRIPT [21], which was also developed in the early 1960s, at the Rand Corporation. Unlike GPSS, SIMSCRIPT is a statement-oriented language. While it is not a direct extension of FORTRAN, the SIMSCRIPT "style" is very similar to that of FORTRAN, and many versions have been implemented by means of a preprocessor which translates the SIMSCRIPT program into FORTRAN.

While GPSS has been developed and improved in many versions, and continues to be in use in the early 1980s, it did not have a major impact on other language developments. I am not aware of any other major discrete simulation languages that have followed the block diagram conceptual approach. (Some of the continuous simulation languages involve block diagrams, but the need for that is more obvious.) The other discrete simulation languages have all tended to follow the statement language orientation, and a number of them have been based on ALGOL rather than a FORTRAN-like approach. In particular, SIMULA [22], which was originally intended as a major simulation language and then became more generally used, was certainly based on ALGOL.

• *APL*

The concepts of APL (A Programming Language) were defined by Kenneth E. Iverson in his 1962 book [23]. At that time he said [24, p. 345]: "The language is based on a consistent unification and extension of existing mathematical notations, and upon a systematic extension of a small set of basic arithmetic and logical operations to vectors, matrices and trees." It is perhaps an understatement to say that APL, as originally described in Iverson's book, was not received by the computing world with much enthusiasm. The combination of an unusual character set and concepts which were quite different from those of the more popular programming languages in the early 1960s tended to make a number of people say, and with some justification, that this was a "notation" rather than a programming language.

Iverson's original motivation was to provide a unifying method of describing and analyzing various concepts in data processing. In [25, p. 662] it is said that: "It is difficult to pinpoint the beginning, but it was probably early 1956," when Iverson was at Harvard. Shortly after Iverson joined IBM in 1960, Adin Falkoff started to work with him, and most of the work done on APL since then has been done jointly.

In the early 1960s, a formal description of the System/360 machine language was undertaken at IBM [26]. This was a significant use of APL, but not the first of its type, since Iverson's book contained a description of the 7090. A running version of a language based on Iverson's notation was developed on the IBM 1620 under the name PAT (Personalized Array Translator) [27], but this was hampered enormously by the need to use standard typewriter characters.

The breakthrough in demonstrating feasibility came with the development of a stand-alone interpretative system on the IBM 360/50 at the IBM Thomas J. Watson Research Center in the mid-1960s. Almost all of the language was implemented; however, the key breakthrough in my own view resulted from two factors. First, a SELECTRIC® printing element was designed to include almost all of the APL characters then in use, and the language which was implemented was in fact then restricted to those characters. Actually not much was left out, and Iverson states [25, p. 665] that some of the necessary changes "were beneficial, and many led to important generalizations." That paper describes several effects that the restriction to a linear 88-character set had on the language, *e.g.*, the notion of composite characters which are formed by striking one basic character over another.

The second breakthrough that helped make APL successful was that the implementation was relatively efficient. At that time (*i.e.*, the mid-1960s) many interactive systems tended to be quite inefficient in execution, partly because they had complex command languages. The ease with which users could communicate with the computer (because APL\360 was a stand-alone system) made it attractive to some users. That "ease of use" factor was something of a technical breakthrough, partly because there was no separate command language.

APL has demonstrated a flexibility for handling different types of problems that has been amazing to those people who view it as being primarily for mathematical problems. As well as by mathematicians and engineers, it has been found to be useful by administrators, secretaries, and people in business environments.

There are strong feelings about APL, both for and against. However, a large number of people who have not been personally trained and introduced to it by the developers have found it to be an extremely useful tool. That pragmatic observation of usefulness would seem to outweigh emotional and even intellectual considerations from language designers and other computer scientists who may or may not feel that it is within the mainstream of language development. Its uniqueness results from (1) its very large and unusual character set and (2) the very large functional capability expressed by individual notations. For example, the notation for multiplication of matrices is shown below for FORTRAN and for APL.

FORTRAN	*APL*
DO 100 I = 1, M	$C \leftarrow B + .xA$
DO 100 J = 1, N	
C(I, J) = 0	
DO 100 K = 1, P	
100 C (I, J) = C(I, J) + A(I, K)*B(K, J)	

From a pure language viewpoint, APL introduced significant new features and concepts. For example, (1) because of the heavy emphasis on vector and matrix operations, there is little need to write the loops required in other languages to achieve equivalent results; (2) it provides a vast number of primitive operators, such as "Floor" and "Minimum"; and (3) the reduction operator allows brevity in applying other operators. However, in spite of (or perhaps because of) its uniqueness, it has had virtually no effect on other language design.

• *PL/I*

Of all the languages developed within IBM, certainly the PL/I effort has to be considered the one with the largest and most grandiose goals. The motivation and background for PL/I stem from two somewhat different sources.

The first involved FORTRAN. By the early 1960s, improvements to the original FORTRAN had been made by delivering FORTRAN II and FORTRAN IV to customers. (FORTRAN III had been only an internal IBM development.) Considerations of extensions to FORTRAN IV were perennially considered. It was clear by the early 1960s that FORTRAN was extremely popular, but would not serve the needs of everyone.

To understand the second viewpoint, it is important to remember that in the early 1960s computer applications (as well as the machines and the programmers) tended to be thought of as either scientific (*i.e.*, engineering and mathematical) or commercial (*i.e.*, business data processing). One of the major objectives of the IBM System/360 was that it would be equally effective for both scientific and commercial applications.

The convergence of these two ideas, *i.e.*, to improve FORTRAN and to create a line of machines to be used across a broad set of applications, caused IBM and SHARE to jointly form the Advanced Language Development Committee of the SHARE FORTRAN project in October 1963, with Bruce Rosenblatt (Standard Oil of California) as Chairman and George Radin as Chairman of the IBM delegation. The group was supposed to specify a programming language which would meet the needs of the user classes indicated above, as well as the needs of systems programmers. Although I do not believe it was stated at that early stage, it was also expected that development of an effective "single" language would be beneficial to both IBM and its customers by eliminating the need for FORTRAN and COBOL.

The earliest significant attempt at a somewhat broad language had been JOVIAL, developed at the System Development Corporation in the early 1960s by Jules Schwartz and others [28]. JOVIAL not only had capabilities for the scientific programmer but also methods for specifying how data were to be allocated within the computer memory; it also introduced in a language the notation of a communications pool (COMPOOL) by which the same descriptions could be used by many programs.

But JOVIAL, which was being used almost exclusively for Air Force projects, did not really satisfy all the needs indicated above. The early orientation of the PL/I design was to provide appropriate extensions to FORTRAN. Although the project was originally referred to as FORTRAN VI, it rapidly became clear that it would be impossible to maintain upward compatibility with FORTRAN IV and also meet the objectives indicated for the design of PL/I.

With regard to the decision to abandon FORTRAN IV as a base, Radin says [29, p. 555]: "In retrospect, I believe the decision was correct. Its major drawback was that, by taking FORTRAN as a base, we could have gone, in an orderly way, from a well-defined language to an enhanced well-defined language. We could have spent our time designing the new features instead of redesigning existing features. By starting over, we were not required to live with many technical compromises, but the task was made much more difficult."

One aspect of the early development of PL/I was the rapidity with which it was supposed to be completed. According to Radin [29, p. 553], from a starting date of October 1963, they "were first informed that the language definition would have to be complete (and frozen) by December 1963. In December we were given until the first week in January 1964 and finally allowed to slip into late February." The objective of this tight time schedule, of course, was to make it feasible to introduce PL/I at the same time as the hardware which became known as the IBM System/360.

There is no space available here to trace the many and somewhat tortuous twists and turns taken to produce the "final PL/I." But PL/I made a number of significant contributions to the technology. For its time, PL/I was the culmination of the procedural line exemplified by ALGOL, COBOL, FORTRAN, JOVIAL, and others. It included almost all the good features from those languages, although generally in a different syntax, and included conceptual features from other languages (*e.g.*, pointers to allow list processing). Although other languages (*e.g.*, ALGOL) contained string variables, PL/I was the first language to provide operations on the strings, such as concatenate. PL/I was the first language to address itself seriously to the problems arising from the need to interact with an operating system. It provided more facilities for dealing with storage allocation, task management, and exception handling than any other language to that date. For example, the user can specify asynchronous execution of tasks and control their execution based on factors such as time delays or completion of another task. PL/I seems to have introduced the concept of generic functions, which means that a single function name can be used to cover a variety of input data types (*e.g.*, fixed or floating point numbers). It used the concept of default conditions very heavily; that eventually was viewed by some people as one of its weaknesses. In its attempt, and indeed its success, at being broad, PL/I became very large and in some cases produced surprising results. For example, many pages in the manual were needed to describe the interaction of implied conversions among variables of different types, and the object code did not always produce the results expected by the programmer's common sense.

PL/I was a major technical undertaking in its language design and implementation. It was the first significant multipurpose language and it introduced a large number of innovations.

5. IBM languages with significant technical impact but not wide usage

IBM developed a number of languages which made significant technical contributions but were not used as much as the ones discussed in Section 4. The most important of these (in my view) are Commercial Translator, FORMAC, QUIKTRAN, CPS, and SCRATCHPAD. Of these, FORMAC had the most effect for reasons to be discussed later. The other four languages had indirect or secondary effects.

- *Commercial Translator*

In order to understand the technical role played by Commercial Translator, it is helpful to understand its place in the environment. Around 1955, Grace Hopper and her department at Remington Rand UNIVAC developed a language for business data processing known originally as B-0 and later renamed FLOW-MATIC. (See description in [3, Chap. V].) They had preliminary specifications as early as January 1955, and the first implemented version distributed to customers was available early in 1958. FLOW-MATIC introduced two major concepts. One was the idea of using relatively long identifiers to get readability (*e.g.*, SOCIALSECURITY, INCOMETAX), along with English verbs for operations such as COUNT, INCREMENT, ADD, etc. The other major concept introduced by FLOW-MATIC was the combination of (1) separating the data description from the statements that were to operate on it and (2) the availability of a fairly flexible data format. Up until that time, all the high level languages had been for scientific computing, in which the types of data needed were integers and floating point numbers, with possible fixed radixpoint, complex numbers, and double precision also being used. But in all of those cases it was assumed that a number representation took an entire machine word (or two) regardless of what its maximum value actually was. Scientific users were not faced with the conceptual problem of data in which some elements might require only one character or even one bit (*e.g.*, distinguishing between male and female) or many computer words (*e.g.*, a person's address). FLOW-MATIC broke new ground in both of those areas, and IBM recognized that it needed to provide a comparable service to its business data processing customers.

As early as January 1958 there were some preliminary specifications for a language (eventually named Commercial Translator) to be used for business data processing; the work was done under the technical leadership of Roy Goldfinger, with Robert Bemer as the manager. Following the philosophy established in FLOW-MATIC, Commercial Translator was an English-like language, but it introduced several significant concepts. One was the use of formulas which were standard in scientific languages but were ostensibly new to the business data processing environment. A second key feature was the introduction of the IF . . . THEN facility, which had first appeared in ALGOL 58. The third idea, building on the data description facilities in FLOW-MATIC, was the concept of allowing several levels of data hierarchy. Finally, the PICTURE clause provided a succinct description of data characteristics such as alphabetic or numeric, placement of the decimal point, number of characters, etc.

But before IBM had implemented Commercial Translator, work on COBOL started outside of IBM. In May 1959 the Short Range Committee was chartered under CODASYL (Committee on Data Systems Languages), which was established under the auspices of the Department of Defense. The Short Range Committee consisted of representatives from six manufacturers, including IBM, and three representatives from government organizations. The primary inputs to the work of the Short Range Committee were FLOW-MATIC, which had actually been in use for over a year, AIMACO (a modification of FLOW-MATIC developed by Air Force Air Material Command), and Commercial Translator [30], which existed as a set of unimplemented specifications. (Brief descriptions of these three languages, and references for them, are in [3, 31]. The history of COBOL is delineated in [31].) The important point is that Commercial Translator was one of the two major inputs to COBOL, and two of the key people on the Short Range Committee which designed COBOL were IBM employees, namely, William Selden and Gertrude Tierney. The significant new concepts of Commercial Translator indicated above were among those included in COBOL. Then, as work on Commercial Translator continued, as the Short Range Committee produced specifications for COBOL, and as Honeywell produced specifications for their own business data processing language known as FACT (see description in [3, Chap. V]), these three languages became interwined. Thus, in 1960 the Commercial Translator manual showed certain concepts that were obtained from COBOL and even from FACT, and, of course, COBOL and FACT were significantly influenced by ideas in Commercial Translator.

Commercial Translator was implemented on the IBM 7070, 7080, and 709/7090. The latter implementation was extremely efficient, which pleased customers, but COBOL prevailed. Thus, after the initial implementations on the 7070 and 7080, IBM discontinued further work on Commercial Translator except for those 7090 customers who insisted on it, and then dropped it completely when going to the System/360. The net result was that Commercial Translator had literally faded from any significant usage by the time the 360 was introduced. For one person's view of IBM's handling of this matter, see [32].

- *FORMAC*

The basic concepts of FORMAC (FORmula MAnipulation Compiler) were first developed by Jean E. Sammet (assisted by Robert G. Tobey) at IBM's Boston Advanced Programming Department in July 1962. I recognized that what was needed was a formal algebraic capability associated with an already existing numeric mathematical language; FORTRAN was the obvious choice. An internal memo describing the basic ideas was written on August 1, 1962, and a complete draft of language specifications was finished in December 1962; implementation design started shortly thereafter. The basic objective was to develop a practical system for performing formal mathematical manipulation on the IBM 7090/94. Originally FORMAC was intended only as an experiment, and there was no plan to make it available outside of IBM. In April 1964 the first complete version was successfully running after extensive testing, and papers on it appeared in the literature [33, 34]. As a result of pressure from numerous people who were interested in trying the system, and also as a way of obtaining feedback from users that would lead to better systems in the future, FORMAC was released in November 1964 but with no committed maintenance or support for it. Nevertheless, there were numerous users.

The use of a computer for performing formal algebraic manipulation went back to 1954, in which two Master's theses (one at MIT and one at Temple University—see references in [3]) involved programs to do formal differentiation. By the early 1960s, various programs were written to do formula manipulation for specialized purposes, *i.e.*, a particular group, such as astronomers or airplane designers, developed a set of routines to do the type of formula manipulation that was needed for their applications. But there was only one attempt at language development, namely ALGY [35], which was an interpretive system on the Philco 2000 computer. It allowed commands such as removing parentheses from an algebraic expression, substituting one or more expressions into another one, factoring a given expression with respect to a single variable. ALGY contained no arithmetic capability, nor was there any facility for loop control or control transfer. The primary objective of FORMAC resulted from the realization that when doing formula manipulation on a computer one needed input/output, numerical arithmetic, loop control, etc. From those needs and premises, the use of a language providing those facilities became necessary, and it seemed logical, as stated earlier, to build on an existing language which already contained the nonformula capabilities. Therefore, FORMAC was literally designed as a language extension of FORTRAN IV and was implemented by a preprocessor which used run-time subroutines to do the formula manipulation.

The conceptual contrast between FORTRAN and FORMAC is shown below.

FORTRAN	*FORMAC*
A = 5 B = 3 C = (A − B)*(A + B) yields C ← 16	C = (A − B)*(A + B) yields C ← $A^2 - B^2$

FORMAC also provided commands for formal differentiation, replacing variables with expressions, removing parentheses, evaluating expressions for specific numerical values, comparing expressions for equivalence or identity, etc. It simplified expressions automatically (as do most of the systems). For further descriptions of the

first FORMAC, see previous citations or [3, Chap. VII]. Eventually a version of FORMAC based on PL/I was developed for use on the System/360 and released, but again with no commitment for maintenance [36]. It was a significant improvement over the earlier version but did not introduce any major new concepts. Both versions were batch oriented but could be used interactively. PL/I-FORMAC remained in minor use in the early 1980s, even though far more sophisticated and better systems existed by then.

During the time that work was being done on the original 7090 FORMAC, a research effort was underway at Carnegie Tech (now Carnegie Mellon University) under the direction of Professor Alan Perlis to develop a system called Formula ALGOL [37]. Both of these efforts proceeded independently; in the very early stages neither was even aware of the other, and after that knowledge did become mutually available, each continued in its own direction. Formula ALGOL was never used extensively outside Carnegie Mellon University. Aside from the obvious differences arising from using ALGOL rather than FORTRAN as a base, there was one major conceptual difference in the two approaches. Formula ALGOL initially used very low level primitives from which the user could build up his own commands, whereas FORMAC used higher level commands.

FORMAC introduced and/or emphasized two major concepts—the desirability of a language for doing formula manipulation, rather than just a series of routines, and the concept of extending an existing language to provide this type of capability. Of the other two systems that appear to be in general use in 1980 one, REDUCE [38], has more or less followed the FORMAC language philosophy by adding capabilities to ALGOL, whereas MACSYMA has developed its own language [39]. Perhaps the most lasting value of FORMAC was its major role, along with Sammet's formation of the ACM Special Interest Group on Symbolic and Algebraic Manipulation (SIGSAM), in getting this technical field started.

- *SCRATCHPAD*

SCRATCHPAD is an experimental, LISP-based, symbolic mathematical system which runs under VM/370 at the IBM Thomas J. Watson Research Center. This work was started by James H. Griesmer in 1965, who was joined shortly after by Richard D. Jenks and later yet by David Y. Y. Yun. Systems outside IBM which were started around that time were REDUCE [38] and MACSYMA [39]. A description of SCRATCHPAD is in [40].

SCRATCHPAD shared with FORMAC (and other systems for symbolic computation) the philosophy that a language (and not just subroutines) was needed for dealing with symbolic mathematical problems. However, it differed from FORMAC in two major ways. One was that a major objective of SCRATCHPAD was to have the language be as natural for mathematicians as possible; thus it was designed *ab initio* and not based on any existing programming language. The SCRATCHPAD language is less procedural, allowing an intended computation to be described by a set of rewrite rules. Although SCRATCHPAD permitted two-dimensional input (for subscripts, superscripts, limits on summations and integrals), the actual input to the computer had to be linearized because standard equipment does not permit two-dimensional inputs; this in my view prevented SCRATCHPAD from meeting the objectives of naturalness to mathematicians.

The second major difference was that SCRATCHPAD was designed from the beginning to be interactive. Unlike normal numerical calculations which can effectively be done in batch mode, for many (although certainly not all) problems involving symbolic computation the user needs to see the formula displayed before knowing what to do next.

SCRATCHPAD provides a wide range of built-in symbolic facilities, which at present are exceeded only by those contained in MACSYMA. SCRATCHPAD facilities include differentiation, integration, polynomial factorization, solution of equations, APL array operations, formal manipulation of finite and infinite sequences and power series, and conversational "backtracking," which allows a user to return to a previous state in his computation.

SCRATCHPAD has never been released outside of IBM, and therefore its technical influence has been limited to an active publication plan in which numerous papers and talks have been given. People outside of IBM have come to the IBM Thomas J. Watson Research Center and successfully used the system for productive work.

- *QUIKTRAN*

Work on QUIKTRAN was started in IBM in 1961 by a group under the direction of John Morrissey. While their original objective was to improve user debugging facilities, this eventually took the form of a dedicated system which was essentially FORTRAN but with powerful debugging and terminal control facilities added. Two major constraints that the designers imposed upon themselves were to use only existing standard equipment (which turned out to be the 7040/44 computers and the 1050 terminal) and to use and stay consistent with an existing language (which turned out to be ANS Basic FORTRAN) defined in [18]. A first version was running in mid-1963. The best language reference is [41]. This system was eventually released for customer use.

The system was designed to handle most legitimate ANS Basic FORTRAN programs. The intent was to allow programs to be debugged using QUIKTRAN and then be compiled for production running on a regular compiler. QUIKTRAN introduced the concept of allowing either the COMMAND or the PROGRAM mode. In the former case each statement entered by the user was executed immediately, and the result was printed at the terminal; this was referred to as the "desk calculator" mode, and the statements were not retained by the system. In the PROGRAM mode, the statements were saved and executed only at the specific request of the user through some other commands. While this concept is very common now, it was either unique or relatively new at that time. JOSS [42] had the same capability, but it is not clear which had it first. In any case both systems were being developed at approximately the same time.

QUIKTRAN was significant from several viewpoints. It was the first on-line system using commercially available equipment (JOSS used the unique JOHNNIAC at the Rand Corporation). QUIKTRAN also retained compatibility with an existing major language and thus made it possible for a user to debug a program on-line and then use a regular FORTRAN compiler for batch production runs. JOSS, of course, and then later BASIC were unique interactive languages designed *ab initio* and each became widely used. There is no clear indication that QUIKTRAN had any long-lasting effect on software technology.

- *CPS (Conversational Programming System)*

CPS was a small, on-line, extended subset of PL/I. It was developed jointly by the Allen-Babcock Corporation (primarily by J. D. Babcock and P. R. DesJardins) and the IBM Corporation under the overall direction of Nathaniel Rochester with significant work by David A. Schroeder. The language is described in [43]; the work started early in 1965, and the initial version became operational in the fall of 1966. The system had two goals: One was to provide a language in the middle area between JOSS and QUIKTRAN. This really meant that it was to be as simple as possible for the terminal user, but the language was to have as much of the syntax of PL/I as possible. The second major objective was to investigate the effectiveness of microprogramming.

The system was originally implemented on a System 360/50 with a special read-only store which was used for special machine instructions to make the language interpreter more efficient. This was an important technical investigation at the time, and appears to be one of the early attempts at implementing software functions in hardware via microprogramming. However, some routines were also written to replace the microprograms and thus avoid the necessity for special hardware. A version of the system (called RUSH for Remote Use of Shared Hardware) was the basis for a commercial time-sharing service offered by the Allen-Babcock Corporation, while the system in use by IBM was called CPS. Starting from the same base the two systems added different facilities and eventually diverged significantly. CPS's primary technical contributions were (1) it was an early interactive subset of PL/I, and (2) some of the translator operations were put into microcode.

6. Formal definition methodology

In addition to specific language developments, there have been two major technical contributions made by IBM employees in the formal definition of programming languages. I think it is fair to say that the first of these may be one of the most significant contributions made to the computing field in general and certainly to the area of programming languages in particular. The two contributions are BNF (Backus-Naur, Backus Normal Form) for describing syntax and VDL (Vienna Definition Language) for defining semantics.

- *Backus-Naur Form (BNF)*

In 1959, John Backus presented a paper [44] in which he introduced to the computing field from the field of linguistics the concept of a metalanguage and showed how it could be used to define the syntax of a programming language, namely ALGOL 58. When the committee to develop ALGOL 60 met and Peter Naur of Denmark was appointed editor of the ALGOL 60 report [45], Naur chose to define ALGOL 60 using this metalanguage, which has been referred to since as "BNF" meaning either Backus-Naur Form (to recognize the contribution of Peter Naur) or Backus Normal Form. The idea was based on the productions of Emil Post, and from a vantage point of twenty years later, it seems very simple. However, the impact has been so profound that it is almost impossible to describe its importance. In my own view, this idea has generated much of the theoretical work in programming languages, and it has the practical advantage of providing a formal way of defining language syntax. The Short Range Committee which developed COBOL was unaware of this work by Backus and developed their own metalanguage in the summer of 1959 (although they called it a notation, not a metalanguage) and used it to define COBOL. (See [31] for a fuller discussion of this issue.) Since 1959, the syntax of most languages has been defined using one or the other of these notations, or a mixture of the two. For example, the metalanguage used in the early major PL/I manual [46] was primarily based on the COBOL metalanguage but contained many elements from BNF. By now BNF has become almost a generic term and, in fact, is sometimes incorrectly used as the name for any metalanguage, whether it is the original one described by Backus or not.

Considerable controversy arose about the use of BNF to describe ALGOL 60 from those people not "in the ALGOL community." Some people immediately regarded it as being an enormous contribution to the rigor of language definition, whereas others found it very difficult to learn and understand. Some people have said that the adoption of ALGOL 60 was severely hampered by the use of BNF; my own view is that its use brought ALGOL into the world in a technically rigorous and understandable fashion and in so doing caused enormous advances in the computing field.

• *Vienna Definition Language (VDL)*

It was recognized that a formal definition of the semantics of a programming language was an even larger problem than formally defining its syntax. (The "syntax" of a language indicates what legal expressions *may be written*, whereas the "semantics" specifies the *meaning* of what has been written. Thus, syntax would indicate that the expression A + B is legal, whereas the semantics would indicate what it meant, since there are other possible meanings than normal addition.)

Apparently the earliest large scale attempt at a formal definition of semantics was the work undertaken in the IBM Vienna Laboratory under the direction of Heinz Zemanek in the mid-1960s. The original ideas of a concrete syntax and an abstract syntax were developed by McCarthy [47], Elgot and Robinson [48], and Landin [49]. VDL (Vienna Definition Language), reduced to its simplest possible terms, simulated an abstract machine and defined the language in terms of the effect that statements in the language would have on this arbitrary machine. For a description of this method, see [50] or the more rigorous [51]. See also [2].

VDL was originally developed for PL/I and then applied to other languages (*e.g.*, ALGOL). At about the same time, work was being done at the IBM Hursley Laboratories to produce a semi-formal definition of PL/I. Some of the flavor of this approach is given in [52]. The use of VDL to define the syntax and semantics of PL/I was the first application of such formalisms to a large and complex language. However, VDL eventually proved too difficult and impractical for compiler writers to use in their development work. Nevertheless, because of the obvious value of a formal semantics definition, the ANSI standard for PL/I [53] was based on the VDL and Hursley approaches; the definition mechanism for the standard PL/I is described in [54]. Various other techniques for formally specifying semantics have been developed (see [55]), but none has received wide practical usage. The major contribution of the VDL effort was to provide a technique for formally defining the syntax and semantics of a complex language and to define PL/I using that technique.

7. Other languages and language activities

IBM has also developed a number of languages which are used in specialized application areas. Examples include COURSEWRITER (for Computer Assisted Instruction), ECAP (for circuit design), and MPSX (for mathematical programming). The whole field of languages for specialized application areas is larger than most people realize—the number of such languages has consistently been about half of all the high level languages developed in the U.S. (see [56–58] for backup data). However, the contributions of IBM in this area do not seem to be nearly as significant as those of the languages cited in earlier sections.

In addition to specific language development and language definition methods, a significant amount of work in compiler optimization has been done. This is being covered in this issue in the paper by F. E. Allen [1].

Summary

IBM and its employees have developed and implemented four major high level languages, APL, FORTRAN, GPSS, and PL/I, each of which made significant contributions to the field. Two languages for formal mathematical computation (FORMAC and SCRATCHPAD) have made important contributions, and two on-line languages (QUIKTRAN and CPS) were innovative at the time they were developed, although they had no long-lasting influence. One business data processing language (Commercial Translator) had a major technical impact via its contributions to COBOL. Numerous languages for specialized application areas have also been developed by IBM. People from IBM contributed to the development of the significant languages developed by interorganizational committees, namely, ALGOL 58, ALGOL 60, and COBOL.

Significant work in the development of formal methods for specifying high level language syntax and semantics was done by IBM employees. The former has had a profound impact on language development, and the latter had some impact on language definition technology.

It should be clear from all of the foregoing material that each of these IBM languages was developed independently of the others. Each built on the technical knowledge available to the developers at the time from within and outside IBM, and some of the individual languages had a progression of named improved versions—specifically APL, FORTRAN, GPSS, and FORMAC. However, the collective set of languages do not form a cohesive technology, because they deal with differing problems in different ways.

Acknowledgments

I would like to thank the following people, each of whom read at least two versions of the full paper: John Backus, Bernard Galler, Charles Gold, and John A. N. Lee. They made valuable suggestions for improvement, and any remaining deficiencies are mine.

References

1. F. E. Allen, "The History of Language Processor Technology in IBM," *IBM J. Res. Develop.* **25**, 535–548 (1981, this issue).
2. P. Lucas, "Formal Semantics of Programming Languages: VDL," *IBM J. Res. Develop.* **25**, 549–561 (1981, this issue).
3. J. E. Sammet, *Programming Languages: History and Fundamentals*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1969.
4. J. Weizenbaum, "Symmetric List Processor," *Commun. ACM* **6**, 524–544 (1963).
5. J. E. Sammet, "Programming Languages: History and Future," *Commun. ACM* **15**, 601–610 (1972).
6. D. Knuth and L. Trabb Pardo, "The Early Development of Programming Languages," *Encyclopedia of Science and Technology*, J. Belzer, A. G. Holzman, and A. Kent, Eds., Vol. 7, Marcel Dekker, Inc., New York, 1977, pp. 419–493. Also in *A History of Computing in the Twentieth Century*, N. Metropolis *et al.*, Eds., Academic Press, Inc., New York, 1980, pp. 197–274.
7. J. W. Backus, "The History of FORTRAN I, II, and III," *History of Programming Languages, ACM Monograph Series*, Academic Press, Inc., New York, 1981, pp. 25–45. Also in *Annals of the History of Computing* **1**, 21–37 (1979).
8. D. T. Ross, "Origins of the APT Language for Automatically Programmed Tools," *History of Programming Languages, ACM Monograph Series*, Academic Press, Inc., New York, 1981, pp. 279–338.
9. A. J. Perlis and K. Samelson, "Preliminary Report—International Algebraic Language," *Commun. ACM* **1**, 8–22 (1958).
10. *History of Programming Languages, ACM Monograph Series*, R. L. Wexelblat, Ed., Academic Press, Inc., New York, 1981.
11. N. Wirth, "PL360, A Programming Language for the 360 Computers," *J. ACM* **15**, 37–74 (1968).
12. "Preliminary ADA Reference Manual (Part A)" and "Rationale for the Design of the ADA Programming Language (Part B)," *ACM SIGPLAN Notices* **14**, No. 6 (1979).
13. *Reference Manual for the ADA Programming Language*, U.S. Department of Defense, Defense Advanced Research Projects Agency, Washington, DC, July 1980.
14. J. W. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Commun. ACM* **21**, 613–641 (1978).
15. J. W. Backus, "The IBM 701 Speedcoding System," *J. ACM* **1**, 4–6 (1954).
16. "Preprints, ACM SIGPLAN History of Programming Languages Conference," *ACM SIGPLAN Notices* **13**, No. 8 (1978). (See also [10].)
17. J. W. Backus *et al.*, "The FORTRAN Automatic Coding System," *AFIPS Conf. Proc., Western Jt. Comput. Conf.* **11**, 188–198 (1957). (Also in *Programming Systems and Languages*, S. Rosen, Ed., McGraw-Hill Book Co., Inc., New York, 1967.)
18. *American National Standard Basic FORTRAN*, ANS X3.9-1966, American National Standards Institute, New York, 1966.
19. *American National Standard FORTRAN*, ANS X3.10-1966, American National Standards Institute, New York, 1966.
20. G. Gordon, "The Development of the General Purpose Simulation System (GPSS)," *History of Programming Languages, ACM Monograph Series*, Academic Press, Inc., New York, 1981, pp. 403–426.
21. H. M. Markowitz, B. Hausner, and H. W. Karr, *SIMSCRIPT—A Simulation Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1963.
22. O.-J. Dahl and K. Nygaard, "SIMULA—An ALGOL-Based Simulation Language," *Commun. ACM* **9**, 671–678 (1966).
23. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York, 1962.
24. K. E. Iverson, "A Programming Language," *AFIPS Conf. Proc., Spring Jt. Comput. Conf.* **21**, 345–351 (1962).
25. A. D. Falkoff and K. E. Iverson, "The Evolution of APL," *History of Programming Languages, ACM Monograph Series*, Academic Press, Inc., New York, 1981, pp. 661–674.
26. A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," *IBM Syst. J.* **3**, 198–262 (1964).
27. H. Hellerman, "Experimental Personalized Array Translator System," *Commun. ACM* **7**, 433–438 (1964).
28. J. Schwartz, "The Development of JOVIAL," *History of Programming Languages, ACM Monograph Series*, Academic Press, Inc., New York, 1981, pp. 203–214.
29. G. Radin, "The Early History of PL/I," *History of Programming Languages, ACM Monograph Series*, Academic Press, Inc., New York, 1981, pp. 551–575.
30. *General Information Manual: IBM Commercial Translator*, Order No. F28-8043, IBM Data Processing Division, White Plains, NY (1960).
31. J. E. Sammet, "The Early History of COBOL," *History of Programming Languages, ACM Monograph Series*, Academic Press, Inc., New York, 1981, pp. 199–243.
32. R. W. Bemer, "A View of the History of COBOL," *Honeywell Computer J.* **5**, 130–135 (1971).
33. E. R. Bond *et al.*, "FORMAC—An Experimental FORmula MAnipulation Compiler," *Proc. 19th National Conference*, Association for Computing Machinery, 1964, pp. K2.1-1–K2.1-11.
34. J. E. Sammet and E. Bond, "Introduction to FORMAC," *IEEE Trans. Electron. Computers* **EC-13**, 386–394 (1964).
35. M. D. Bernick, E. D. Callender, and J. R. Sanford, "ALGY—An Algebraic Manipulation Program," *AFIPS Conf. Proc., Western Jt. Comput. Conf.* **19**, 389–392 (1961).
36. *PL/I-FORMAC Interpreter*, IBM Corporation, Contributed Program Library #360D 03. 3.004, 1967, available through IBM branch offices.
37. A. J. Perlis and R. Iturriaga, "An Extension to ALGOL for Manipulating Formulae," *Commun. ACM* **7**, 127–130 (1964).
38. A. C. Hearn, "REDUCE 2, A System and Language for Algebraic Manipulation," *Proc. Second Symposium on Symbolic and Algebraic Manipulation*, Association for Computing Machinery, New York, March 1971.
39. J. Moses, "MACSYMA—The Fifth Year," *Proc. Eurosam Conf., ACM SIGSAM Bull.* **8**, 105–110 (1974).
40. R. D. Jenks, "The SCRATCHPAD Language," *Proceedings of the Symposium on Very High Level Languages, ACM SIGPLAN Notices* **9**, 101–111 (1974).
41. T. M. Dunn and J. H. Morrissey, "Remote Computing: An Experimental System, Part 1: External Specifications," *AFIPS Conf. Proc., Spring Jt. Comput. Conf.* **25**, 413–423 (1964).
42. J. C. Shaw, "JOSS: A Designer's View of an Experimental On-Line Computing System," *AFIPS Conf. Proc., Fall Jt. Comput. Conf.* **26**, Part 1, 455–464 (1964).
43. *Conversational Programming System*, IBM Corporation, Contributed Program Library #360D 03. 4. 016, 1967, available through IBM branch offices.
44. J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-

this area is Backus-Naur Form—a notation for describing the syntax of programming languages. This is reviewed briefly by Sammet in her paper. Although no generally accepted method for formally defining the meaning of programming languages exists, a basic method has been developed at the IBM Laboratory Vienna, called the Vienna Definition Language (VDL). It has been applied to a large, complex language, PL/I. One of its developers, Lucas, reviews the method and compares it to successor approaches.

Throughout the history of data processing, workers have sought methods to model computing system performance, including that of its software support, so as to predict the performance of planned systems, to optimize the performance of existing systems, and to gain insights into complex system interdependencies. Two extensively published participants in this activity, Bard and Sauer, review the history, theory, and application of computer performance modeling, with particular emphasis on the contributions of IBM.

The general purpose programming support dealt with in most of the papers in this chapter only provides the soil in which application programs designed to solve specific problems can grow. IBM has participated rather directly in this aspect of computer science as well. As a sample of this activity, we include a paper by Flatt, who has spent much of his professional career dealing with large-scale scientific computations. His paper reviews the progress made in modeling complex energy- and environment-related phenomena, including studies of some of the consequences of energy consumption on the environment.

Editor

M. A. Auslander
D. C. Larkin
A. L. Scherr

The Evolution of the MVS Operating System

The mechanization of computer operations and the extension of hardware functions are seen as the basic purposes of an operating system. An operating system must fulfill those purposes while providing stability and continuity to its users. Starting with the data processing environment of twenty-five years ago, this paper describes the forces that led to the development of the OS/360 system design and then traces the evolution which led to today's MVS system.

Introduction

Computer operating systems first began to appear twenty-five years ago. Since then, an operating system discipline, complete with new terminology, new employment categories, large expenditures for research and development, and formal academic training, has evolved.

In this paper we review the evolution of operating systems in IBM, drawing primarily on our experience with the Multiple Virtual Storage (MVS) operating system [1] and its progenitors OS/360 and OS/370 [2]. [Another paper in this issue reviews the development of the Virtual Machine Facility/370 (VM/370 operating system) [3].] In the next section of this paper we recall the environment that prevailed throughout the past quarter century, showing some of the major changes in technology and applications. Then the major areas of operating system function are discussed in terms of the significant technological advances made in them. Finally, we consider current trends and likely future directions.

The fundamental purpose of operating systems is to facilitate the use of computer systems. Functions provided can be grouped into two categories: (1) automation of computer operations; and (2) extensions of hardware function.

The earliest systems automated operations by mechanizing inter-job transitions [4]. Jobs were executed sequentially, one at a time. In contemporary systems, separate jobs often exist simultaneously as interactive applications, with the attendant complexity of allocating resources. Modern systems provide automatic resource allocation and tuning to aid computer administrators in the scheduling of work and the balancing of resources for multiple applications.

The extension of hardware functions probably started with the symbolic assembler [5]. Other examples include higher level languages, such as FORTRAN, COBOL, BASIC, etc., and assembler macros to perform higher level arithmetic functions and input/output operations. Over the years, significant amounts of software have been produced to raise the level of the interface to hardware so that application programmers would not have to deal with such details as timing, hardware geometry, and error recovery, and could deal instead with macro- rather than micro-level operations. One aspect of this trend has been to mask the application programmer from the details of I/O devices and other hardware elements, so that hardware conversion could be done without requiring changes in application programs.

Another major source of function has been the movement of facilities common to many applications into the operating system. The earliest examples of this were the large card tubs of subroutines (*e.g.*, square root) that appeared in many machine rooms in the 1950s. These functions are first seen in new applications. Later, as they become more generalized and more popular, they find their way into the operating system. For instance, the

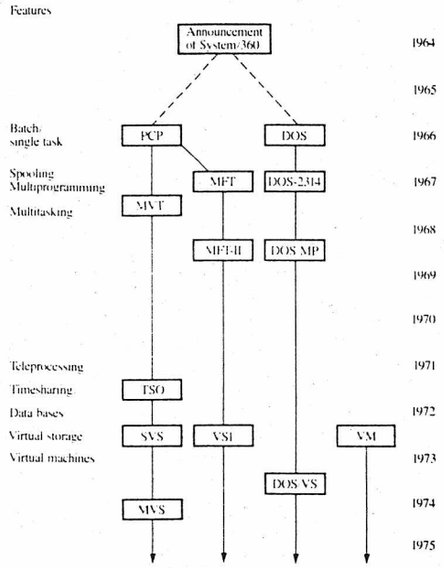


Figure 1 The evolution of OS/360, where PCP = primary control program, MFT = multiprogramming with a fixed number of tasks, MVT = multiprogramming with a variable number of tasks, TSO = time sharing option, SVS = single virtual storage, MVS = multiple virtual storage, DOS = disk operating system, and VM = virtual machine facility.

hierarchical data organization supported by IBM's IMS-DL/I program product was invented initially to support bill-of-materials processing in a manufacturing application of the late 1960s. As this support was generalized, it became clear that its usefulness ranged far beyond the manufacturing industry, and today DL/I is one of the most widely used methods for organizing data bases.

The environment: 1956–1981

To convey the environment in which designers of operating systems function, we first look at computing as it was done twenty-five years ago, contrasting it to that of today. In 1957 the total computer processing power installed in the U.S. was about ten million instructions per second. Today's installed capacity in the U.S. is three to four orders of magnitude greater. This tremendous growth of computing power led directly to the need for and definition of operating systems.

Another way to view the progress made since 1956 is to look at the size of a software system and how it has changed over the period. For instance, the FORTRAN Monitor System (FMS) and its programming libraries, widely used on the IBM 704/7090, fit on as little as one reel of tape. During application program execution, FMS required only the low 100 memory locations to contain the programming for job-to-job transition. The size of FMS itself was about 10 000 binary cards of programming, approximately one million bytes. This contrasts with today's MVS system, which arrives stored on 17 tapes that are the equivalent of 13 million cards, approximately 520 million bytes.

Over the last twenty-five years, IBM designers and programmers have created over twenty major, separate families of general purpose operating systems. The systems culminating in MVS are shown in Fig. 1 and their capabilities in Table 1. Generally these have been motivated by unique hardware, but in recent times IBM has developed different operating systems for the same hardware systems. For instance, on the latest IBM System/370 computers, there are eight IBM operating systems in use. These are: Disk Operating System/Virtual Storage Extended (DOS/VSE), Operating System/Virtual Storage 1 (OS/VS1), OS/VS2 (now called Multiple Virtual Storage or MVS), VM/370, Airlines Control Program (ACP), and Time Sharing System/370 (TSS/370), with continued substantial use of the System/360 Operating Systems, OS/360 and DOS [6].

Even within an operating system, diversity pervades. At least six different telecommunication access packages have been offered for OS/360: BTAM, QTAM, TCAM, RTAM, XTAM, VTAM [7]. In addition, several users have developed their own.

While we have created a large number of programming products over the last twenty-five years, only a relatively small number have faded from existence. An extreme example of the longevity of these products is that programming and hardware shipped as recently as 1980 still provides the capability to execute programs originally written to run on the IBM 650 and 1401 computers, first shipped in 1954 and 1960, respectively. A more meaningful example is that today's MVS systems must be capable of executing many of the programs that were written to run on the 1966 OS/360, Release 1. One of the most challenging aspects of providing software for IBM users is the huge investment in application programs for IBM computers, and one of the primary requirements placed on any operating system is to continue to provide the capability to successfully use these programs. Thus, designers of IBM operating systems have been faced with the need to provide for extensions of function while at the same time preserving the ability of existing interfaces to operate the same way they had in the past.

Table 1 Capabilities of MVS and its predecessors.

Generation	*Operations*	*Extensions of function*	
		Hardware functions	*Application functions*
Pre-operating system (early 1950s) with, *e.g.*, the 701	Manual (*e.g.*, each job step required manual intervention) No multiple-application environment support	Symbolic assembler Linking loader	Subroutine libraries in card tubs, manual retrieval
First generation (late 1950s and early 1960s) with, *e.g.*, FMS, IBSYS on the IBM 704, 709, and 7094	Automatic job batching Manual device allocation, setup, work load scheduling No multiple-application environment support Off-line peripheral operations	Higher level languages—FORTRAN, COBOL Primitive data access services with error recovery	Subroutine libraries on tape, automatic retrieval Primitive program overlay support
Second generation (late 1960s) with, *e.g.*, OS/360 on System/360	Multiprogramming Primitive work load management Primitive tuning (device, core allocation) Spooling, remote job entry Operator begins to be driven by the system Primitive application protection Initial multiprocessing (loosely and tightly coupled)	More higher level languages—PL/I, ALGOL, APL, BASIC Device independence in data access First random access data organizations Primitive software error recovery, full hardware ERP's Array of hardware function extensions Supervisor call routines	DASD subroutine libraries Full facilities for programmed overlays Interactive program development support Primitive automatic debugging aids First application subsystems Checkpoint/restart
Third generation with, *e.g.*, MVS OS/VS on System/370	Integrated multiprocessing (loosely and tightly coupled) Work load management extensions More self-tuning, integrated measurement facilities Less operator decision making, fewer manual operations Full interapplication protection, data and program authorization Primitive storage hierarchies for data	Virtual storage Device independence extended Hardware error recovery extended to CPU, channels Operating system functions begin to migrate to hardware	Growing libraries Overlay techniques obsoleted by virtual storage Symbolic debugging aids Primitive data independence Integration of application subsystems Software error recovery for system and applications

However, there are exceptions. For instance, in DPPX [8], one of the operating systems for the IBM 8100 Distributed Processing System, compatibility requirements could be relaxed because DPPX was aimed at new application environments. In spite of this, DPPX pays homage to the past in that it has COBOL and FORTRAN and must be able to interchange programs and data with other systems.

In addition to the need to provide continuity with the past, the other major constraints dictating design trade-offs in operating systems are the capacities and speeds of the hardware for which the software is being designed. These two factors, compatibility with the past and hardware parameters, constitute the major reasons for maintaining separate operating system designs in contemporary systems. The occasional unique function in a particu-

Table 2 Numbers and varieties of devices that can be included in an MVS system.

Disk magnetic storage units	10 types
Drum magnetic storage units	1 "
Mass storage system	1 "
Diskette	1 "
Magnetic tape	8 "
Card readers/punches	5 "
Paper tape reader punch	1 "
Printers	10 "
Optical character recognition unit	5 "
Magnetic ink character recognition unit	1 "
Operator's console	7 "
Telecommunications control units	6 "
Display terminals	7 "
Keyboard printer terminals	4 "
Remote high-speed terminals	6 "
Other terminals (including other processors)	30 "

This table does not depict all actual different model numbers; rather, it gives the number of different devices of each type.

lar system (*e.g.*, the virtual machine facility in VM/370) generally exists because of the relative cost of providing it in other systems compared to its benefits, rather than technical feasibility.

Generally, software has adapted to the rapidly changing circumstances of the data processing industry. This includes new application types, new hardware variations, and new styles of usage. On the other hand, change has been heavily dependent on the past because of the large investment all of us have in maintaining the ability to run existing programs. Over the years what has happened can be viewed as a process of natural selection, similar to other evolutionary processes.

Another dimension of the problem of designing operating systems is the diversity of hardware configurations that must be supported by a single operating system. MVS, for example, runs on eleven different processing units, some of which allow symmetric and some asymmetric two-way multiprocessing. The performance range supported is well over an order of magnitude. The minimum MVS production system has two megabytes of main storage and as few as four disk drives. Conversely, one large MVS configuration in the United States has seven interconnected processing units in a single room. This configuration has more than 450 direct access devices and services approximately 15 000 terminals in more than 50 locations. The total instruction execution capacity is over 40 million instructions per second.

The largest processor currently supported by MVS has a real main storage capacity of 32 megabytes and a maximum of 1023 devices connected to its I/O channels. Anywhere from two to 16 I/O channels are supported.

Table 2 shows the numbers of devices of various types that can be selected for inclusion in an MVS system. Providing for this large number of combinations influences the way operating systems are designed. The I/O configuration can change dramatically from installation to installation and, in fact, from month to month. Of course, it would be impractical to include all of the programming necessary to run all of these devices in every system. Moreover, as desirable as device interchangeability would seem to be, it ought to be possible to exploit new technology and new hardware capabilities. For these reasons, operating systems like MVS are designed to be modular, with support for particular device types selectable for a particular installation.

A final problem for IBM operating systems designers is the number of installations using each system. Changes and extensions made to our systems are, soon after release, experiencing extremes in work loads and varieties of applications. A change that is incorrect in the most obscure special case will soon generate a problem report or "APAR." A modification to an undocumented and unguaranteed internal characteristic of an interface is often exposed by applications which work only because of that characteristic. This necessarily contributes to a design approach that is conservative and evolutionary and that groups modifications so that extensive tests can be performed on modified systems before their shipment. Yet greater stability is one of the most asked for features by customers of our products.

In summary, our experience of developing operating systems is that of responding to requirements in the basic functional areas under the constraints of a wide range of hardware configurations, large numbers of installations, and compatibility. The following sections explore the approaches we have taken in response to these motivating requirements.

Operations

The earliest use of computers was characterized by totally manual operations; that is, the sequence of programs to be run was controlled manually, media containing needed data were mounted and dismounted by the operator, usually following written instructions. A typical compile-load-go sequence in the 1950s was accomplished by the operator placing the binary card deck for the compiler in a card reader, with a program to be compiled behind it, and pressing the Load Cards button on the console. The compiler assumed that certain "scratch" tapes were available and used these during the compilation. At the end of the compilation, output was printed, and cards representing the object program were punched. This I/O activity was not overlapped with any other activity. The operator would then pull the object deck out of the card punch, walk over to the card tub, and select the required subroutines and an eight-card linking loader that was placed at the front of the resulting deck. These cards were then placed in the system card reader, and the operator once again pressed the Load Cards button. If any data were required by the program, the appropriate media would be prepared by the operator.

Generally, if operations were any more complex than described, written instructions were provided for the operator, or more likely the programmer would be personally present. "Programmer present" runs were so common that a space on the job card was provided to indicate that type of run. In these cases, the operator would telephone the programmer when his run was about to be made.

Accounting was done on a total system basis, often using a time clock to log a user on and off of the system. Users and operators took great pride in the speed with which they could mount tapes and operate the hardware to minimize the idle time between jobs.

By the late 1950s, this type of operation had been semi-automated, with the card tub of library subroutines placed on tape and the linking loader having the capability of searching this library tape for unresolved external references. Moreover, the ability to change from the compilation to the execution phase of the job without manual intervention had been provided. Still, communication with the operator was done for each program in pretty much an *ad hoc* way. The 704 had a series of lights on the console called Sense Lights, and often instructions to the operator had statements like "if Sense Light 4 lights, turn on Sense Switch 3."

Through the early 1960s, systems were characterized by manual device allocation, setup, and work load scheduling. Multiprogramming was not provided, and only a single application would run at a time. It had been recognized very early that the printing of output and the reading and punching of cards were operations that could not be performed efficiently on a high-speed central system. Thus, tape input and output operations were substituted, and the card-to-tape, tape-to-printer, and tape-to-card operations were done off line. At first this was accomplished with special purpose equipment designed specifically for these functions. Then these off-line operations were done with small computers, such as the IBM 1401. Later the IBM 7040 was linked via a high speed connection to the larger IBM 7094 to do these operations and related job scheduling functions. This "Direct Coupled System" is one of the earliest examples of automated job scheduling and setup. It significantly improved the utilization of the controlled systems by overlapping setup and peripheral input and output with system operations. This approach survived into the System/360 era as the Attached Support Processor (ASP) package and is now embodied in the JES3 subsystem of MVS.

It was not until the multiprogramming support provided in the mid-1960s that job scheduling and peripheral operations were done simultaneously with other work on the same processor [9]. It was at this time that the operator, rather than having control of the system, began to be provided with instructions by the system. Comprehensive job control languages were introduced to allow the application programmer to completely specify sequences of job steps, as well as device and data requirements for each, so that the system could allocate and sequence the appropriate programs and resources. The OS/360 Job Control Language (JCL) was widely regarded as complex and difficult to use. Yet its contribution was to separate application programs from such operational considerations as device types, addresses, sequences, conditions, and timing of execution. Thus, it was possible to create new applications by writing JCL programs to apply existing processing and utility programs in new ways.

Today, the most significant operational challenges remaining are the mechanization of the remaining manual functions and the improvement of existing algorithms for work load management so as to react effectively to changing work loads and to the lack of availability of particular pieces of hardware or other resources due to errors, failures, or contention. In the future these decisions will be increasingly pre-programmed into the system, and more of the manual media operations will be eliminated through the use of storage hierarchies and automated media handling techniques.

- *Maintenance and service*

Large numbers and sizes of programs are typically involved in the modern data processing installation. Moreover, the problems of fixing errors, installing new versions of software packages, updating critical data, etc., must all be done in the face of the need to operate more continuously. In the early 1960s, it became apparent that *ad hoc* techniques would no longer suffice. By the 1970s several data base oriented systems had been created in IBM to keep track of modifications, error fixes, and versions of programs. Because code modifications made to fix errors sometimes themselves contain errors, such a system must allow swift recovery if such errors occur. Thus, if a modification contains an error, it must be possible to restore the system to its original state very

rapidly. With the large installations of today, and especially with the distributed systems that are beginning to appear, it is essential to provide the ability to manage the servicing of large numbers of systems with a single centralized package. Thus, record keeping and distribution facilities must be expanded to allow for tracking up to hundreds of systems.

Since the time to diagnose problems is important, and since often problems encountered in a system at one installation have been found and fixed earlier at others, it is useful for an installation to have access to a central data base of problem symptoms and fixes. During the 1960s this was accomplished by distributing this information on listings. Recently, remote interactive access has been provided to this data, and, in the future, symptom descriptions to be used to search these data bases will be generated automatically.

- *Resource allocation*

OS/360 was the first general purpose IBM operating system to provide for complete sharing of hardware resources. The major resources considered for shared usage in the original design were the processor, its memory, direct-access storage devices (disks, drums), I/O devices (tapes, card equipment, printers, etc.), and the space on individual disks and drums.

OS/360 provided for processor sharing by introducing the notion of a "task" [9]. An OS/360 task is an implementation of what computer scientists eventually came to call a process. An additional characteristic of a task in OS/360 is that resources, including other tasks, are allocated to a task. Further, tasks are designed to allow full multiprogramming of their execution. The OS/360 designers recognized that operating system functions should usually contend for resources by being themselves treated as tasks.

The task structure was originally intended to support not only long running batch jobs but also the short running, response-oriented processing associated with transaction and telecommunications environments [9]. Unfortunately, the generality of the task concept was too great to allow for efficiency. The fact that the task is interruptable by other higher priority tasks and is the anchor for resource accounting, allocation, and recovery required significant processing just to create a task. Thus, the overhead was impractical for very short-lived pieces of work. A widely used technique to overcome this problem was to have a task waiting for a signal to restart, rather than creating a new one each time. This technique created long queues of mostly dormant tasks that had to be searched, and it added overhead in other areas. Also, because the task did not fully support all of its characteristics in its first implementations, early application subsystems like QTAM, CICS, and IMS [10] provided their own substitutes for the task. In MVS, a new construct called the System Request Block (SRB) was created to be purely a unit of work for the processor. Interruption ability was not provided, and no other resources were anchored to the SRB. Thus, an SRB in MVS can be created and given control of the processor using a trivial number of instructions, compared to the thousands required to create a task.

The original intent in the OS/360 design was to allocate memory to each task on demand [9, 11]. Memory would then be held by the task until it was either explicitly released by the task or the task terminated. The existence of this interface, and the crucial and new requirement that programs ask the operating system for memory space and accept that space at whatever location it was provided, made for some early difficulties. However, this interface once and for all solved the problem of how the operating system and the application program can share the processor memory, even though they are independently designed. For this reason alone, similar storage allocation mechanisms persist in all operating systems today.

From the beginning, the designers of OS/360 were concerned with the possibility of a resource allocation deadlock [12]. They chose the philosophy of complete deadlock prevention by predetermining the order of allocation: I/O devices, data sets, memory, then the processor. It was not recognized until late in the design that the earlier assumption of complete dynamic allocation of main memory from a common space would lead to deadlocks. To overcome this problem, the application programmer had to specify the aggregate storage requirement of his job in advance. This "region" could then be allocated, and the possibility of a deadlock during execution was avoided.

By the late 1960s, OS/360 was being severely strained by two emerging requirements. The under-utilization of memory caused by region allocation was becoming ever more troublesome. At the same time, it was recognized that application development could be eased if programs did not have to be fitted into the smallest possible address space. Dynamic address translation hardware and demand paging techniques were applied to these problems throughout the 1960s in such IBM systems as the 7044 and System/360 models 40 and 67 [3, 11]. In 1970, demand paging was added to OS/360. In MVS, a large number of regions can be created, with the size of each region the same and usually not a factor to be considered by the programmer. The technique of demand paging is used to share real memory among the existing regions. Because demand paging allows dynamic reallocation of memory, deadlocks cannot arise.

The second emerging requirement was for time sharing [13]. This technique can improve the capabilities of program developers and problem solvers by allowing continuous communication with the computer. Time sharing was first provided by a number of special purpose systems in the early 1960s. By 1967 there were over 30 different special purpose time sharing packages running or planned for System/360 hardware. The Time Sharing Option (TSO) was intended to integrate time sharing functions into the OS/360 base. However, the batch allocation of devices and data space present in the base were inappropriate. In practice, time sharing is workable when the storage devices in use are permanently mounted and when space on them is dynamically allocated. To accomplish time sharing in OS/370, dynamic disk space management interfaces were added to the existing batch mechanisms. It is noteworthy that TSO uses essentially the same interfaces and mechanisms of OS/360 for purposes for which they were not initially intended.

Time sharing was implemented in TSO without address translation hardware. Instead it used a memory swapping technique pioneered at MIT on the IBM 7094 [14] in the early 1960s. The fundamental resource management problems of time sharing, for which these special systems had previously been developed, can be dealt with by the multiprogramming and paging mechanisms of MVS. In creating MVS from the OS/360 and TSO componentry, many of MVS's major structures came out of generalizations of techniques introduced in TSO.

Today, three resource allocation issues concern operating system developers. The first is the impact of multiple processors. It might seem that a multiprogramming operating system could be converted to multiprocessing with minimum difficulty. In practice, however, an operating system contains many internal information resources which are frequently locked and unlocked. As the number of processors increases, so does the contention for these internal resources with the attendant loss of performance when conflicts arise. To control such conflicts, MVS uses carefully designed lock structures to minimize the cost of locking an available resource and the probability of needing an unavailable resource. The MVS approach is based upon the experience gained during the 1960s with the OS/360 and TSS support for multiprocessing. The OS/360 structure had a single lock and produced serious degradation in many important environments. The TSS approach used myriad locks but in an uncontrolled sequence that created deadlock and recovery difficulties.

The other two resource allocation problems deal with resources that were not shared when OS/360 was first designed: communication facilities and data stored at the logical record level [15]. The former is the subject of the Systems Network Architecture (SNA) [16] and has required substantial innovation. The second is primarily a problem of data base subsystems [17].

- *Security and integrity*

Over the years, operating systems have played an increasingly important role in providing tools to allow an installation to protect itself against unauthorized access to data or other computer facilities [18]. In the earliest systems, little or no provision was made to protect against such use. In the early 1950s, before the advent of multiprogramming, such protection was generally not needed, since a job's data were loaded onto the system with the job and taken down at its completion. When jobs were first batched on an input tape, there was the ever-present danger of one job forward or back spacing the input tape, causing other jobs to be skipped or repeated. At some universities, this was an annual occurrence as new programming students "tested the system." It was not until late in the 1950s that IBM computers began to provide protection. There was a multiprogramming special feature for the 7090 that provided for privileged instructions—only programs in certain states could execute instructions that performed I/O operations and other management tasks.

In the System/360, the separation of privileged instructions from those intended for use by application programs was made quite clear; it was theoretically possible for software to prevent unauthorized use of any function. The original direction taken in OS/360, however, was that deliberate penetration attempts would not be prevented; that is, the system was designed only to prevent casual or accidental penetration or unauthorized use. It was not until the late 1960s that it became apparent that this philosophy was inadequate. Operating systems had to prevent a program from gaining access to data, services, or other facilities that it was not authorized to use. MVS was the first IBM system for which a systematic attempt had been made to eliminate all exposures to unauthorized use and for which a commitment had been made to correct any such exposures found. MVS had a number of authorization schemes by which particular users or programs could be allowed or denied the use of certain data and functions.

Since the introduction of MVS in 1974, additional loopholes have been identified and closed. Also, more elaborate authorization capabilities have been provided, as well as detection capabilities, to facilitate the identifi-

cation of a perpetrator, and hardware assistance, such as encryption devices [19].

Extensions of hardware function

The following subsections deal with extensions to hardware function to enable application programmers to more easily exploit computer systems. The first subsection deals with extensions that raise the level of function of the processor's instruction set. The second deals with similar extensions in the I/O subsystem area. The final two subsections deal with error recovery, which could be viewed as the creation of ideal computer elements that apparently never fail.

- *High level languages*

Early operating systems, such as the FORTRAN Monitor System, were based on an intimate connection between their language processors and the system which managed compilation, loading, and execution of programs written in those languages. An important advance of OS/360 was to separate the compiler from the operating system. A compiler is, from the operating system view, just another user program which takes input and produces output. Whether or not the user then decides to have the program loader load and execute that output is unimportant. The only tie between the compiler and the system is the conventions for representing machine language programs as operating system files.

This separation has a number of advantages. Compilers can be offered and maintained independently of the operating system. New languages or new compilers can be developed easily. Various organizations inside and outside IBM can offer languages and compilers which do not depend, for their success, on operating system modifications. This has led to a healthy growth in both languages and compilers. Finally, this decoupling encourages general purpose operating system interfaces. However, the danger is that significant functions may not be available to higher level language programmers.

Another aspect of the interaction between high level languages and operating systems is the implementation language of the system. Until the late 1960s, IBM software was written in macro assembly language. It was believed that compilers could not produce code efficient enough for operating systems. While this assertion was true in 1970, the reduction in programming errors and the increase in program extensibility resulting from the use of high level languages offset the space and execution time penalties. By 1980, most programmers could not have consistently exceeded the efficiency of compiler optimized code. Now essentially all new operating system code is written in a high level language.

- *Access methods*

As operating systems evolved in IBM, they were driven by a number of forces towards the development of generalizations of their hardware input/output devices. The primary forces operating were:

Ease of programming Programs must often deal with the mechanical complexities of the device, variations in the storage medium, complicated procedures, and error detection and recovery strategies. These often change with each new hardware innovation.

Device independence For purposes of data storage, the important distinctions from the point of view of the application programmer are record size, retrieval order, selection techniques, etc., rather than media dependence. A program written to process a sequential stream of 80-character records should be the same no matter where the data are stored.

Data integrity The shared use of direct access storage device (DASD) space and the shared (often read-only) use of data require some trusted program between the application and actual manipulation of the media.

Concurrent operation The involvement of the operating system in every input/output operation makes possible the incorporation in its implementation of strategies for overlapping computation and input/output activity without requiring that individual programs include complex code (such as double buffering) to produce local concurrency. The concurrency arises rather because one job is computing while another is doing input/output operations.

Early operating systems often dealt with the simpler aspects of some of these needs. Beyond that, the libraries for some high level languages contained input/output routines which eased the programming burden for users of that language. With the introduction of OS/360, the notions of an access method and of a data set organization were introduced in a uniform way [7].

Data set organizations are abstractions that expose to the application program those distinctions which it must deal with and shield it from other device distinctions. The original data set organizations of OS/360 were sequential, partitioned, indexed sequential, and direct access.

Access methods are collections of operations that can be applied to read and write data in these organizations. The original access methods were Basic Sequential (BSAM), Queued Sequential (QSAM), Basic Partitioned (BPAM), Indexed Sequential (ISAM), and Basic Direct (BDAM). The distinction between basic and queued had to do with whether or not the access method supported implicit blocking and deblocking internally, as well as whether or not the access method was capable of a read-ahead input/output strategy. In hindsight, one could argue that this distinction was short sighted. However, in 1963, it was difficult to decide to isolate the programmer from the device at all, let alone to impose a blocked, buffered interface as the only one available. Real concerns with performance and a confusion of the technique of buffering with the function of sequential access both contributed to this viewpoint.

These small sets of organizations and access methods served for access to all devices and data during the early history of OS/360 and OS/370. Changes in this area have been remarkably rare, considering that these interfaces were simply invented and implemented all at once by the system architects. The major changes to occur in the intervening years reflect a new requirement and the recognition of a new technology.

The new requirement is the desire to attain uniformity and device independence for terminals. BTAM, TCAM, and VTAM represent three attempts at communications access methods. A number of reasons exist for this relative lack of success in standardizing communications access methods.

First, the communication devices did not become important until after the introduction of OS/360. Without a body of experience, and with dramatic and fundamental changes in hardware capability (from teletypewriters to CRT display terminals and beyond), it was difficult to foresee all future needs in designing an interface. Beyond that, more than in any other input/output area, the application program must deal with the details of the display device. Adding more bytes to a disk track can easily be hidden from the application program using that disk to store information, but adding more rows or columns to a display necessarily affects the application design decisions relating to the human interface that the display represents. Providing for terminal device independence in applications is one of the remaining challenges that will get much future attention.

The other important change was a response to a new technology. The indexed sequential access method was designed to support data organizations in which contents retrieval (keyed records) was used to maintain the appearance of a sorted file while allowing efficient random update and retrieval. When it was designed, it was commonly believed that the best way to do such retrieval was to exploit the key search capability of the DASD hardware. After the introduction of OS/360, the use of tree structured, balanced indexes (B-trees) as a technique for managing content addressing was developed, and it became clear that this organization was superior to the ISAM approach. To offer this new technology, IBM introduced VSAM, an access method which in some sense provides all the organization and retrieval functions of the other access methods. Again, history and the need for continuity tie us to using and supporting the new and the old. While the use of ISAM is waning, the use of the sequential, partitioned, and direct access methods continues and will continue for the forseeable future.

The OS/360 access methods went a long way towards achieving ease of programming, device independence, data integrity, and concurrent operation. As much as any other feature of OS/360, they have made it possible for programs written fifteen years ago to continue to operate on today's totally new and often quite different hardware and operating systems.

In OS/360, and subsequently in MVS, the format of recorded data and the available operations differ by access method and organization. Thus, device dependence was, to some extent, replaced by data set organization dependence. For example, sequentially organized data cannot be accessed by record number without a format conversion. It is possible, as is demonstrated by the DPPX operating system and by CMS, to achieve uniformity and provide better usability as a result.

- *Hardware error recovery*

IBM's first computers were shipped with little more programming than that used for hardware error diagnosis. Typically, engineers develop such programs for the early models of a system. As computer programs became more sophisticated, these diagnostics became less effective. By the end of the 1950s, it was commonplace for a nontransient error to be undetected by the manufacturer-supplied diagnostics but to show up readily when trying to execute application software. It was during this period that the gathering of hardware error status and statistics was first introduced into IBM software. During the early 1960s, the software was designed to stop upon the detection of any hardware error, under the assumption that to continue would produce unpredictable results. This philosophy was reinforced by the fact that the hardware often signaled a status that was undefined. Nevertheless, elaborate algorithms were created to re-try operations if errors might be transient or, in the case of input/output functions, to attempt alternate paths to devices. For instance, if a control unit for disks had paths to two I/O channels, one of the strategies to recover from a channel failure would be to attempt access through the second channel.

It was not until the early multiprocessing systems that serious attempts to recover from central processor errors

were made. The IBM 9020 System [20], created to assist in air traffic control, was a fully redundant multiprocessing system that could continue despite the failure of any single element. Much of the effectiveness of the software in accomplishing this goal was due to the nature of the application. Since the major function of the system was to process incoming radar data, little stored data had a useful life of more than a few seconds. Thus, it was possible to reconstruct the state of the application a relatively short time after a failure.

The general purpose systems did not fare quite as well. It was not uncommon when a single element failed, even though spare hardware was available, for the software to be unable to untangle itself well enough to continue. By the late 1960s it was apparent that a major undertaking was required to address the effectiveness of software in recovering from hardware errors and, indeed, hardware failures. If this were successful, a two-processor system could reliably continue with one processor remaining. One aspect of this work involved a joint effort by hardware architects and software designers to ensure that adequate status information was presented so the software could continue from a known state. In MVS, this type of recovery was implemented on a large scale within the operating system. Thus, in the event of failure of a single processor in a two-processor system, the software can move the work being done on the failed processor over to the remaining one and simulate two processors until a pre-defined state has been reached. Subsequently, the system can operate as if it were a normal single-processor system.

- *Software error recovery*

One of the lessons learned during the original implementation of MVS was that software errors manifested themselves in more complex and unpredictable ways than hardware errors. Because the relative frequencies of hardware and software errors were comparable, it was decided to attempt to handle them both with a single set of facilities that would signal the program in control at the time of the error. The signal took the form of an interrupt to a pre-specified exit program. If no such exit program had been specified, then control would be given to an exit specified by the next higher level program (*i.e.*, the program that called the program in control). Since the operating system itself is the ultimate caller of every program, ultimately a recovery exit would be reached.

The general ground rule in the design of MVS was that every part of the operating system should be designed to include such recovery facilities. The job of each recovery exit was to assess any damage and either repair it or remove ongoing work from the system. In some cases it was deemed acceptable to lose resources for the duration of the program or job that was running or until the next time the system was initialized. For instance, if an error destroyed the records of free storage and if those records had to be reconstructed but some were lost, then perhaps several thousand bytes of storage might be unavailable for a period as a consequence of the recovery action. This is preferable to losing the entire system and having to reinitialize it.

The MVS programs associated with error exits are called functional recovery routines (FRRs). As would be expected, these were more effective in dealing with hardware errors than software errors. Because they were executed only when an error was detected, their effectiveness could really be assessed only over time and usage in real situations. Over the years this experience has led to increasing the effectiveness of these routines, and MVS is a more robust system as a consequence of having them. A remaining challenge is to substantially improve the effectiveness of detecting software errors.

The difficulties in creating an effective recovery program increase with the generality of the program for which recovery is being attempted. For example, if a failure occurs during a basic supervisor service, such as establishing a connection to a file (*e.g.*, OPEN), and if in fact that data set cannot be made available to the application program, the specific actions to be then taken are really best left to the discretion of the application program. Depending on which file cannot be accessed and its importance to the application, the program may or may not be able to continue. Thus, the burden could be shifted entirely to the application programmer.

In the interests of application programmer productivity, a great deal of effort has gone into the error recovery facilities of the data base and data communications subsystems of operating systems. For instance, the designers of the IMS system [10, 15] have gone to extraordinary lengths to protect the integrity of data under its control from errors introduced by both hardware and software failures. The concept of a transaction has been incorporated into IMS, and the system is designed so that the data base reflects changes made only by successfully completed transactions. Thus, if a program performing the work of a transaction fails, no updates to the data base are ever actually reflected. To implement this concept, IMS maintains journals of changes and undoes any changes made by a failing transaction program. In addition, IMS can maintain journals of data base changes so that if a disk file is lost due to an error, recovery can be accomplished by loading a check-pointed version of the data onto the disk and by processing the journal of changes against it, bringing it up to date. Transaction-oriented recovery has been implemented on a variety of systems to some degree, including CICS and DPPX/DTMS. Over the past ten years, the concept of transaction-oriented data base recovery has been generalized and extended. It is now accepted as a standard way to deal with this type of data processing and error recovery.

A future trend in this area will certainly be the combining of all the techniques so as to provide more effective and rapid recovery in the event of a hardware or software failure. As systems get larger, restart times are increased, making faster recovery more critical. Also, the significance of failures is aggravated because systems are being directly used more by human beings. Thus, the effective use of standby equipment, allowing switch-over in seconds, will become more important.

- *Application subsystems*

The earliest application subsystems were implemented to provide batch remote job entry (RJE), improve batch job scheduling, and provide interactive facilities for time sharing and other problem solving applications. These were implemented on the earliest versions of OS/360 in the middle 1960s. HASP and ASP are two examples of the RJE and batch job scheduling subsystems. APL [21] on OS/360 is an example of the latter type. As the techniques pioneered in these application subsystems became better understood, they were integrated into the operating systems, and use of the separate subsystems gradually faded.

The early 1970s also saw the popularity of the data base/data communications (DB/DC) systems. These systems used different forms of resource allocation to achieve goals similar to those of the time sharing systems. The chief difference between the DB/DC systems and the time sharing systems was that the former generally did not need to maintain program context between input messages. That is, an incoming message could be processed and the data bases updated without the need to maintain program variables until the arrival of the next input. On the other hand, interactions with users in a time sharing environment were more conversational, and systems generally had to provide the ability to relate successive inputs and programs needed to maintain continuity from one input to the next. This difference led to a number of DB/DC resource allocation strategies which provided for higher efficiency than was available in time sharing systems. This improved efficiency was significant enough to warrant special purpose implementations. Here again, usage during the 1970s gravitated toward two or three packages for the System/360, and in the late 1970s the movement of functions from the subsystems into the operating system base had begun to occur.

The overall trend in this area is that, when operating systems do not schedule resources or provide similar functions in a way which is adequate for new functions, these new functions are provided in application subsystems. As these functions are identified it is expected that they will be integrated into subsystems and then the subsystems themselves into operating systems. Thus, a natural selection process identifies the soundest techniques and the most generally acceptable solutions.

Conclusions

The most unusual aspect of the IBM programming environment is undoubtedly the combination of forces brought to bear by the large number of installations and their diversity of applications. This, coupled with the ongoing stream of new hardware devices, has created a situation in which small changes are amplified substantially. In other words, the law of large numbers is operative, and it is easy to see the results of any change, both positive and negative. The rate of change has been immense and progress has been made in virtually every dimension of the computer programming craft.

Over the years, there has been substantial movement of function out of the manual realm, as well as the realm of the applications programmer, into the operating system itself. The operating system has taken over functions relating to increasing the usability of the hardware, to protecting application programs from changes in hardware, and to saving them from the need for conversion in the event of such shifts. Operational considerations have changed from the stand-alone single batch systems of the 1950s and early 1960s to the multi-system, geographically dispersed, but centrally managed, complexes of today, with the attendant operational and control requirements of such systems being placed in the various operating systems. The evolution of increasingly complex and capable data organizations, including the relational facilities that are now being used, represents another thrust of increasing function. Other trends include more facilities to ensure automatic error recovery and data integrity in the event of failures.

In all cases the effect has been to reduce the need for application programs to provide various functions. The net effect is that, compared to twenty-five years ago, the average application program as written by the programmer represents a much smaller fraction of the cycles executed by the hardware itself. In 1956, it was indeed rare that IBM written software took more than a few cycles of the system. Today it is not at all uncommon for IBM provided software to use 90 percent or more of the capacity of the system, with the user written application programs using the remainder.

Other strides during the period include decreasing dramatically the rate at which errors are introduced into programming. In the last ten years there is evidence to suggest an order of magnitude improvement in the number of errors per line of code introduced in IBM operating systems. Moreover, major efforts have been made to reduce the impact of errors by providing various recovery strategies in the software itself. Also, provisions are being made to allow for continuous operation, whereby changes and updates can be made to a system without stopping it. Finally, ease of use from the installation of the system to its actual use has lately received significant attention.

The overall increases in the capability of the operating system are really the result of what is perhaps the major economic trend of the last twenty-five years in the data processing industry—the steady decrease in the cost of data processing hardware in the face of steady increases in the cost of human labor. Thus, every year it becomes easier to justify automating manual functions through the use of computers and of augmenting those functions already automated to make them more efficient in their use of human time. We have lately seen the second and third implementations of applications that were originally created in the 1950s as batch programs. For instance, in 1956 many computer users were beginning to automate inventory control through the use of transactions written on sheets of paper, transcribed to punch cards, sorted, and processed in a batch mode against a master inventory file. Today such applications are implemented as interactive DB/DC programs that update inventory records on a real-time basis. Obviously, there is an incremental value to this type of operation, since inventories are maintained on an up-to-the-minute basis, and other aspects of operations can also be automated.

At the moment there seems to be no end in sight for this trend. It is anticipated that more and more emphasis will be placed on the traditional operating system areas of further expediting operations, extending hardware functions, and providing common application functions as part of operating systems. As was the case twenty-five years ago, it is apparent that the requirements of the next twenty-five years are not all known today, and the ability of our programs to be adaptable to unforeseen requirements will remain a very important characteristic. Implementing functions in a generalized way is the best preparation for unforeseen requirements.

Acknowledgment

The authors would like to acknowledge Mr. Bob O. Evans, IBM Vice President—Engineering, Programming and Technology, for stimulating the creation of this paper.

References

1. A. L. Scherr, "Functional Structure of IBM Virtual Storage Operating Systems, Part III: OS/VS2-2 Concepts and Philosophies," *IBM Syst. J.* **12**, 382–400 (1973).
2. G. H. Mealy, "The Functional Structure of OS/360, Part I: Introductory Survey," *IBM Syst. J.* **5**, 3–11 (1966).
3. R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Res. Develop.* **25**, 483–490 (1981, this issue).
4. A. S. Noble, Jr., "Design of an Integrated Programming and Operating System, Part I: System Considerations and the Monitor," *IBM Syst. J.* **2**, 153–161 (1963).
5. Nathaniel Rochester, "Symbolic Programming," *IRE Trans. Electron. Computers* **EC-2**, 10–15 (1953).
6. G. Bender, D. N. Freeman, and J. D. Smith, "Function and Design of DOS/360 and TOS/360," *IBM Syst. J.* **6**, 2–21 (1967).
7. W. A. Clark, "The Functional Structure of OS/360, Part III: Data Management," *IBM Syst. J.* **5**, 30–51 (1966).
8. S. C. Kiely, "An Operating System for Distributed Processing—DPPX," *IBM Syst. J.* **18**, 507–525 (1979).
9. B. I. Witt, "The Functional Structure of OS/360, Part II: Job and Task Management," *IBM Syst. J.* **5**, 12–29 (1966).
10. W. C. McGee, "The Information Management System IMS/VS—Part I: General Structure and Operation," *IBM Syst. J.* **16**, 84–95 (1977).
11. L. A. Belady, R. P. Parmelee, and C. A. Scalzi, "The IBM History of Memory Management Technology," *IBM J. Res. Develop.* **25**, 491–503 (1981, this issue).
12. J. W. Havender, "Avoiding Deadlock in Multitasking Systems," *IBM Syst. J.* **7**, 74–84 (1968).
13. Allan Scherr, "An Analysis of Time-Shared Computer Systems," Ph.D. Thesis, MIT Press, Massachusetts Institute of Technology, Cambridge, MA, 1967.
14. F. J. Corbató *et al.*, *The Compatible Time-Sharing System, A Programmer's Guide*, MIT Press, Cambridge, MA, 1963.
15. W. C. McGee, "Data Base Technology," *IBM J. Res. Develop.* **25**, 505–519 (1981, this issue).
16. J. H. McFadyen, "Systems Network Architecture: An Overview," *IBM Syst. J.* **15**, 4–23 (1976).
17. M. W. Blasgen *et al.*, "System R: An Architectural Overview," *IBM Syst. J.* **20**, 41–62 (1981).
18. C. R. Attanasio, P. W. Markstein, and R. J. Phillips, "Penetrating an Operating System: A Study of VM/370 Integrity," *IBM Syst. J.* **15**, 102–116 (1976).
19. W. F. Ehrsam, S. M. Matyas, C. H. Meyer, and W. L. Tuchman, "A Cryptographic Key Management Scheme for Implementing the Data Encryption Standard," *IBM Syst. J.* **17**, 106–125 (1978).
20. G. R. Blakeney, L. F. Cudney, and C. R. Eickhorn, "An Application-Oriented Multiprocessing System: II—Design Characteristics of the 9020 System," *IBM Syst. J.* **6**, 80–94 (1967).
21. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York, 1962.

Received July 6, 1981; revised July 27, 1981

M. A. Auslander is located at IBM Corporate Headquarters, Old Orchard Road, Armonk, New York 10504; D. C. Larkin and A. L. Scherr are located at the IBM System Communications Division laboratory, Neighborhood Road, Kingston, New York 12401.

R. J. Creasy

The Origin of the VM/370 Time-Sharing System

VM/370 is an operating system which provides its multiple users with seemingly separate and independent IBM System/370 computing systems. These virtual machines are simulated using IBM System/370 hardware and have its same architecture. In addition, VM/370 provides a single-user interactive system for personal computing and a computer network system for information interchange among interconnected machines. VM/370 evolved from an experimental operating system designed and built over fifteen years ago. This paper reviews the historical environment, design influences, and goals which shaped the original system.

Introduction

The Virtual Machine Facility/370, VM/370 for short, is a convenient name for three different operating systems: the Control Program (CP), the Conversational Monitor System (CMS), and the Remote Spooling and Communications Subsystem (RSCS). Together they form a general purpose tool for the delivery of the computing resources of the IBM System/370 machines to a wide variety of people and computers. The CP and CMS components evolved directly from an experimental operating system designed and built by the author and his group at the IBM Systems Research and Development Center in the mid-1960s. This center, now called the IBM Scientific Center, is located in Cambridge, Massachusetts.

CP is an operating system that uses a computing machine to simulate multiple copies of the machine itself. These copies, or virtual machines, are each controlled like the real machine by their own operating systems. CMS is an operating system that supports the interactive use of a computing machine by one person. It is the typical operating system used within each virtual machine. RSCS is the operating system used to provide information transfer among machines linked with communications facilities. These three systems, used together, produce a general purpose multiple access facility. Other operating systems can be used on each virtual machine as well and might be selected for batched job processing, for compatible interchange with other systems, or for other purposes.

In the spirit of reviewing the twenty-five years spanned by the *IBM Journal of Research and Development*, we take a moment to discuss the historical environment, design influences, and goals which formed the first ancestor of VM/370. The paper is historical and is not intended to be a critical look at operating system design. (That subject scarcely lacks coverage; see, for example, [1].) A few aspects of the CP/CMS design merit special attention, but mostly it is the particular selection, combination, and implementation of features which have proved useful. This retrospective look may provide some insight into the personality, capability, and potential of the VM/370 design.

Information can be found elsewhere to describe the current incarnation of VM/370 and related subjects. The manuals [2] provided for this IBM product cover concepts, user commands, and system information. More generally, virtual memory concepts [3], virtual machine concepts [4], and the virtual machine environment of VM/370 [5] have been discussed at length. The bibliographies of these references are very good.

An historical perspective

The roots of VM/370 are most deeply entwined with the style of use of the computing machines of the 1950s by scientists and engineers. In those days, the machines were used as personal tools, much like their predecessors which had been designed and dedicated to specific appli-

cations. Unlike the earlier machines, however, they did not sit idle when a problem was solved; their architecture was more general, with easily changed stored programs directing their actions on a variety of problems. These programs, hand crafted for the slow and costly hardware, required the entire machine. The users were normally present to make sure things were going well and, if not, reacted in real time to correct things or to collect information useful for diagnosis. The running time of a program was quite long compared to these user actions and to the time required to set up the machine for the next problem.

As machines became faster and program execution times shortened, the overhead of human operation became more significant. Simple job monitor systems were introduced to reduce the cost of the idle time between execution of different programs. These systems terminated a program in a preselected way, prepared the machine, and began execution of the next job. Machines continued to grow in capability and speed and to decline in cost per computing unit. With larger memories and independent I/O operation came the possibility of more efficient machine utilization. A portion of the machine could be dedicated to the programs which assisted in machine operation, the operating system. As a result, overall productivity of the system increased. Peripheral computers were used to prepare job streams for the main computer; they printed and punched batched output as well. The turnaround time of a job from submission to completion was measured in hours, sometimes days. The prescheduling of jobs resulted in more efficient machine utilization, but the users were more isolated from the machine. Developing and testing programs was a frustrating experience.

Thus the stage was set for the direct interactive use of machines by some users. The machines, occupied gainfully by the steady stream of batched work, could be interrupted now and then by people demanding a small amount of computer time. It was thought that the costs of such time-shared use could be reduced to the point where people, using typewriter-like equipment, could again command the machine to do their bidding. The pioneering work in such a general purpose time-sharing system was done at M.I.T. in the early 1960s [6]. With this system, called the Compatible Time-Sharing System (CTSS), a normal batched job stream was run as background to keep the computer busy. At the same time, several users could enter commands to prepare, execute, and terminate their programs. The machine directly responded to each of them in real-time. Programs, data, and textual material were created, modified, executed, formatted, and the like in a style similar to that produced today by VM/370, a style which is now common to many personal computers as well.

Batched job systems remained the main mode of computer operation and continued in their evolution. They could overlap the termination, operation, and setup of sequential jobs. Later, more complex systems would multiplex the execution of several programs in an effort to reduce the idle time of individual machine components. For all but the most specialized of uses, like airline scheduling or real-time control, these sophisticated batched job operating systems managed the machine well.

During this same period, time-sharing systems were designed to solve similar problems; these solutions recognized the on-line user as an important new factor. In addition, user interfaces, tools, and application programs were developed uniquely for interactive use. Techniques to support dynamic allocation of the machine and demand scheduling became more efficient and predictable. Most importantly, the machine and people costs steadily moved to favor interactive systems. Today's batch operating systems include modifications and extensions to support interactive use, although the style and capability presented to the user tend to reflect the heritage of the underlying system.

The future promises inexpensive and powerful machines that can individually serve each of us as assistants as a machine now stores, modifies, and formats this text under my command. A pattern of use, similar to that decades ago, is emerging with the use of personal computers as machines dedicated to a single user. The cost of these modern machines is so incredibly low compared with their progenitors that the idle time caused by human interaction or lack of work is not significant. But these machines will not be isolated in laboratories. They will be interconnected in many ways as the emphasis moves from computing convenience to information access.

Computing use has come full circle to repeat history in a variety of ways. The ingredients of dedicated machine use and interactive support, occurring separately in the past for most people, will combine with machine to machine communication to characterize these new systems. It is precisely because these features also characterize VM/370 to its users that this system design should be viewed in the modern context. VM/370, with its collection of interconnected, independent machines each serving one user, can provide architectural compatibility with the future's networks of personal computers.

Design influences and goals

The CP/CMS system was conceived in 1964 as a second-generation time-sharing system for the newly announced IBM System/360. It began as an experimental software project [7, 8] designed to support all the activities of the Cambridge center, including such diverse activities as operating system research, application development, and report preparation by programmers, scientists, secretaries, and managers. Soon after its inception, it was convenient for the system to be recognized and financially supported from outside of the center as a tool to evaluate and test the performance of operating systems. It later gained acceptance as a time-sharing system after its installation at M.I.T.'s Lincoln Laboratory. Originally called a pseudo-machine time-sharing system, CP/CMS was named a virtual machine system from the description of similar but independent ideas [9]. It was to integrate traditional computer operations within a more general and interactive framework.

In 1966, CP-40 and CMS both became operational using an IBM System/360 Model 40 modified by the addition of an associative memory device for dynamic address translation [10, 11]. CP-40 was run only on this unique hardware at Cambridge. At about the same time, CP-67 was built to use the address translation feature of the newly announced System/360 Model 67. CP-67 and CMS were installed on many of these production machines. VM/370 became available in 1972 for the IBM System/370 computer family whose members all included virtual memory hardware. CP and CMS have seen continuous production use since 1967 with over 2500 systems now in operation. We now review some of the work which influenced the conception and design of the CP/CMS system.

- *Systems influence*

The early 1960s witnessed many concurrent projects in time-sharing system design. In addition to the first operational general purpose systems at M.I.T. and at Bolt, Beranek, and Newman, several of the early efforts mentioned in [12] took place at the Carnegie Institute of Technology, IBM Corporation, M.I.T., Rand Corporation, System Development Corporation, and the University of California at Berkeley. A wide variety of ideas and opinions covered every aspect of the sought-after interactive facilities, from the overall system architecture to the details of program interfaces, especially editors. Many days were spent discussing issues, such as character by character input processing, which today command just as much attention as then.

Recall that very little hardware was available for continuous and convenient human interaction with machines. Terminals were a problem to find and attach to a computer. Special purpose hardware was usually required, and primitive connections, such as driving a teletypewriter from a memory bit, were not unknown. In addition, the computers in those days did not provide many of the protection features commonplace today. Sharing a machine safely among programs and users provided many thorny problems. As one solution, most early time-sharing systems provided new or modified languages operating through interpreters or restricted execution contexts.

The Compatible Time-Sharing System [6, 12], first operational in 1961 and in production use at M.I.T. from 1964–1974, most strongly influenced the CP/CMS system design. CTSS, a first-generation time-sharing system, provided a subset of the machine for use by normal batch programs. The compatibility and protection techniques it used were simple and effective. The background programs were run without modification as with a normal system. The time-sharing supervisor would steal and restore the machine without their knowledge. This technique was extended to its fullest in CP/CMS. Many other CTSS design elements and system facilities, like the user interface, terminal control, disk file system, and attachment of other computers, served as operational prototypes.

The implementation of CTSS illustrated the necessity of modular design for system evolution. Although successful as a production system, the interconnections and dependencies of its supervisor design made extension and change difficult. A key concept of the CP/CMS design was the bifurcation of computer resource management and user support. In effect, the integrated design was split into CP and CMS. CP solved the problem of multiple use by providing separate computing environments at the machine instruction level for each user. CMS then provided single user service unencumbered by the problems of sharing, allocation, and protection. As an aside, the MULTICS system [13] of M.I.T.'s Project MAC and CP/CMS were both second-generation systems drawing heavily on the CTSS experience with very different architectural results.

Recursive systems, such as biological reproduction and to some extent the early LISP design [14], exhibit powerful, elegant concepts which influenced the CP/CMS system design. In reproduction, the cell is duplicated including the duplication mechanism. With LISP, several primitive functions act on a simple data structure to define an architecture within which all functions and data, both system and user, are represented. The duality of functional mechanisms and their representation as the data upon which they operate are illustrated by these systems and by CP/CMS.

- *Hardware influence*

The family concept of the IBM System/360, announced in spring 1964, was a most amazing turning point in comput-

er development, one which was not universally greeted with enthusiasm. We believed that the architecture of System/360, combining scientific and commercial instruction sets, would be around for a significant period of time. The trauma associated with widespread recoding of programs also pointed to a long life. In addition, we speculated that many operating systems and a large number of application programs would be produced over the lifetime of that machine design.

How could we design a single operating system to accommodate all of these programs in a smooth and compatible way? What interface and capabilities could be designed to support such a wide range of applications? Going back to basics turned up the answer: The instruction set, the essence of this new machine line as documented by the System/360 principles of operation manual, would be the interface to the time-sharing control program. Each user would have the complete capability of the System/360. Machine sharing and allocation would be accomplished below this level by the Control Program. This apparent replication of the machine for each user not only guaranteed compatibility but provided an avenue for future machine development whereby resource control could be incorporated into the machine architecture.

The design of System/360, in order to facilitate the multiplexed execution of several jobs in a scheduled job environment, provided two instruction execution states: privileged and problem. The instructions available in problem state are those commonly used by application programs. They are innocuous to other programs within the same machine and can be safely executed. However, privileged instructions affect the entire machine as well as report its status. As they are encountered in problem state, the machine blocks their execution and transfers control to a designated program. When using CP, each virtual machine program is actually executed in problem state. The effects of privileged instructions are reproduced by CP within the virtual machines.

The System/360 machine design made possible the reasonable implementation of multiple machine environments with only one major exception. There was no practical way to move programs within the memory after they had been prepared for execution. Yet experience with demand scheduled systems suggested the need for dynamic program relocation. The IBM 7094, modified for CTSS, contained a relocation register to offset memory addresses, but it was of little use because a program could only be moved as a unit. Some technique had to be found to break programs into pieces which could be moved into, out of, and within the memory independently of each other.

- *Virtual memory influence*

In the late 1950s, work at the University of Manchester led to the Ferranti Atlas machine design, which provided automatic extension of main memory using drum storage [15]. Motivated by the high cost of fast memory and by the inconvenience of machine-specific two-level store manipulation, it was to spawn virtual memory. The memory was split into fixed length page frames which were used, as required, to execute a program stored in the larger address space of the drum. This idea was extended [16] and incorporated into the 1964 machine design proposal of M.I.T.'s Project MAC for the MULTICS system. Multiple memory segments, each like a single memory broken into pages, formed the program address space. The same idea was developed [17] and implemented by IBM in an experimental system called the M44/44X, first operational in 1965. The memory architecture of the System/360 Model 67 and the System/370 family follow the same design trend.

The hardware modification to the IBM System/360 Model 40 used for the CP/CMS project, most similar to the Atlas design, provided for relocation and protection. The main memory was split into a fixed number of pages. During program execution, each memory address and an active user number were compared, using an associative array, to the list of virtual page addresses present in real memory. The position of the matching entry, if it existed, was encoded into a physical page address. No degradation of the memory cycle was required for this action. If no match was found, a page fault was indicated, resulting in a machine interrupt. The memory structure provided by segments and pages, key to the MULTICS design for example, was not required by the CP/CMS design. A difficulty of virtual memory, solved by programming and some restriction, was the control of System/360 I/O channels—processing units operating independently of the dynamic address translation hardware.

- *Design goals*

The early time-sharing systems supported input, edit, and output of programs and data; most supported programs in machine code, like the output of the FORTRAN compiler; some only interpreted programs in source language form; but no system could safely execute machine code which like itself manipulated all features of the system. The ability of these systems to support the full interactive cycle of program refinement, test operation, production use, and program enhancement for a wide range of applications was not extended to the operating system itself. System developers could only reap the benefit of computer assisted programming up to the point of actually trying the code, at which time they required a dedicated hardware system, usually at night or on weekends. We wanted all hardware function delivered to the user, if desired. In addition, the system was designed for continuous operation. CP/CMS was the first system to extend availability to all users.

The necessity of compatibility for evolutionary growth of software was demonstrated by CTSS; for hardware, by the IBM System/360 family. The ability to accomplish steady growth, accommodate change, and provide smooth access to past knowledge is a requirement for any successful system architecture in the future. This includes the ability to utilize production batch, real-time, and other specialized systems as the hardware speed and cost allow. It also includes the capability to incorporate more sophisticated hardware construction techniques. For example, a piece of CP might be implemented within hardware to extend machine capability. Users of virtual machines and their operating systems would see no change in this function except possibly in cost or speed. Conversely, CP might simulate a new hardware feature to be tested. Performance might be poor, but programs using this feature could be run. This was done within IBM to prepare System/370 programs using a System/360.

A virtual machine cannot be compromised by the operation of any other virtual machine. It provides a private, secure, and reliable computing environment for its users, distinct and isolated like today's personal computer. Experiments by one user present no problems to a payroll job on another virtual machine. New facilities, such as a data base system or specialized device support, can be added without modification or disruption of current capabilities. Modular functionality is strongly fixed in the CP/CMS design. Each virtual machine system can provide an important service: by CMS, for execution of user programs; by RSCS, for a store and forward computer networking system. Many important areas remain to be considered. Those capabilities necessary within CP to support virtual machine operation can be carefully chosen and implemented; unessential function can be moved into a virtual machine.

One of our important design goals was the production of a virtual machine identical to its real counterpart. We expected movement of systems among real and virtual machines. For example, system device addresses, fixed on real machines, were easily changed on virtual machines. A program tailored for a real machine could be accommodated simply on a virtual machine. This level of indirection also made the virtual machines insensitive to many changes in the actual hardware. Some devices can even be simulated efficiently. It is tempting to produce features unique to the virtual machine. But any schism between virtual and real machine capabilities can only limit future options. Features available within the virtual domain can be evaluated as extensions to the machine architecture, then incorporated or eliminated. CMS originally operated on real hardware but that is no longer possible without recoding.

The design of CP/CMS by a small and varied software research and development group for its own use and support was, in retrospect, a very important consideration. It was to provide a system for the new IBM System/360 hardware. It was for experimenting with time-sharing system design. It was not part of a formal product development. Schedules and budgets, plans and performance goals did not have to be met. It drew heavily on past experience. New features were not suggested before old ones were completed or understood. It was not supposed to be all things to all people. We did what we thought was best within reasonable bounds. We also expected to redo the system at least once after we got it going. For most of the group, it was meant to be a learning experience. Efficiency was specifically excluded as a software design goal, although it was always considered. We did not know if the system would be of practical use to us, let alone anyone else. In January 1965, after starting work on the system, it became apparent from presentations to outside groups that the system would be controversial. This is still true today.

The control program

A program coded for a computing machine can produce results in either of two ways. It can be loaded into the machine and executed. Or a person can interpret the program, step by step, using the architecture manual as a guide. In both cases the results should be the same. (Actually, results may differ because timing relations among machine components or error conditions may not be precisely defined.) In the first case the program is being executed on a real machine; in the second case, we can say the program is being executed on a virtual machine. A computer can be programmed to replace the person in this interpretation process. The Control Program is an operating system which uses this technique to determine the operation of virtual machines.

The Control Program is a general multiprogramming system that uses virtual machines to organize independent job streams. A portion of the machine controlled by CP is needed to support the allocation and management of the rest of the machine. The remainder is available for the virtual machines. The techniques used by CP to share machine components range from the simple to the sophisticated. For example, a magnetic tape drive is attached, if not in use, to a virtual machine with a simple CP command. It is dedicated to that machine until detached.

Each virtual machine is identified by an entry in the CP directory. This entry contains, among other things, the definition of permanent virtual disks mapped to specified disk volumes. Temporary disks, available to an active machine for a session, are dynamically defined by a CP command from space set aside for that purpose. The simulation of a virtual card reader and punch or a line printer is more difficult. All information to and from these devices is stored by CP in its spool system. It is labeled with the virtual machine identification and associated with its correct virtual component. CP operates the real equipment consistent with the user expectation of a dedicated resource. Control of the memory, CPU, and I/O channels requires more sophisticated methods. Management of the machine resources to achieve various response and efficiency profiles for each virtual machine is the most difficult problem to solve. This is because of the wide variety of programs accommodated by CP, ranging from conversational interactions to day-long computations. The constantly varying workload blurs the traditional distinction between batch and time-shared operation.

All work done by the machine under control of CP is the result of virtual machine action. There is no other facility, like a background job stream, to be used in place of a virtual machine. Any virtual machine defined in the CP directory can be activated by a person using a keyboard and display terminal. The terminal is then used as the control console of a virtual machine. Commands to CP take the place of switches and buttons used to start and stop the CPU, display and store memory contents, trace execution, etc. It is also possible to command CP to create or delete hardware components, to change the machine configuration, or to perform other services, like the selection of printer paper. Commands to CP can be thought of as requests to a machine operator of extraordinary capability. In addition to console function mode, the user's terminal can be used by the program being executed in the virtual machine. This is the normal method of communication between the user and application programs. But another type of terminal connection is possible. In this case, a virtual machine is not activated by the user. Instead, with a CP command, the terminal is attached to a selected machine that is already active. The terminal is then under exclusive control of the program in that machine. A multi-user information management system might be such a program.

The System/370 virtual machine can be in basic or extended control mode. Basic control does not include the virtual memory hardware. This type of machine is used by CMS and other operating systems that do not themselves utilize the address translation hardware. Extended control mode is selected when an operating system that controls virtual memory, such as CP or OS/VS, is executed in a virtual machine. This might be the case when a new version of CP is to be tested. At many installations, a batch operating system is run in a virtual machine to be compatible with traditional procedures, application programs, or other locations. In fact, at a few installations, CP is used only to concurrently operate production and test versions of the same batch operating system. Virtual machines are not provided for personal computing, at least not outside of the operations group!

A key feature of the CP/CMS design is the independence of each virtual machine. All connections between virtual machines are explicit because they are specified and controlled by CP, whether via shared memory or disks, I/O channels, unit record media, or telecommunications lines. There are no hidden dependencies or relationships between systems in separate virtual machines. If these machine interconnections are within the domain of the computer architecture, evolutionary growth from virtual to real systems is possible. As larger, faster, and less expensive machines become available, the software systems supporting interconnected virtual machines can move smoothly to collections of real machines.

Any computer, real or virtual, without software is of use only to computer students. This is the case with a newly activated virtual machine. Of course, instructions can be stored in the memory, one by one, using the terminal as the machine console, and then executed. No operating system is needed. This is impractical but certainly familiar to those with today's microcomputers or yesteryear's memories. As with a personal computer, a choice of many operating systems may be available to the user. A most important characteristic of the Control Program is to give each user the choice of any software, whether backlevel, standard, modified, or home-brewed. Normally the machine console is first used to specify an initial program load operation. The program so loaded is usually an operating system, probably CMS, that will operate the computer in a manner convenient to the user.

Conversational monitor system

In the beginning, the initials CMS had various meanings: Console Monitor System, Cambridge Monitor System, and so on. By any name, CMS is a disk-file-oriented operating system to support the personal use of a dedicated computer. It is a single user system providing function in the style of CTSS. CMS began operation on a real IBM System/360 serving the user at the system console. When CP became operational, CMS moved from the real to the virtual hardware.

The heart of CMS is the file system, which manages permanent information stored on disk. The user of the file system, via CMS services, sees a collection of named files, grouped by disk. Each file contains records of fixed or variable length. Any record can be accessed, but additional records must be appended to the file. Each disk managed by the file system contains a set of files, all listed in a single-level directory. These files and their directory are completely resident on one disk and are independent of the files on other disks. The file system automatically manages the physical organization of the data. It stores all files in fixed length data blocks, maintains pointers, and allocates space invisible to the file system user, who only uses disk name, file name, and record number to access data. Each disk is managed by its user, who must create, archive, and erase files as necessary.

Sharing of data among users is accomplished outside of CMS by connections established by CP between virtual machines. Disks shared among machines provide the usual means of accessing common information. Typically, several disks, in addition to each user's private disk, contain programs, data, and the CMS operating system. This structure of multiple disks, each with a single directory, was chosen to be simple but useful. Multi-level linked directories, with files stored in common areas, had been the design trend when we began. We simplified the design of this and other components of CMS to reduce implementation complexity. We had to produce a system we could use to produce more sophisticated systems. We did not have the people and time to solve the problems associated with more ambitious features. In addition to disk sharing for multiple user file access, copies of data files are frequently transmitted between users via simulated unit record equipment. This method generalizes nicely for users of connected collections of machines and is discussed in the RSCS section.

The CMS nucleus resides in low memory. Although it is key-protected to prevent accidental destruction of important data like file directories, user programs have the run of the machine and can modify the system or use privileged instructions. Of course the integrity of the other machines is not compromised. Programs are coded in some source language, processed to produce machine instructions, and loaded into memory. At that point, they can be executed or stored on disk as a memory image, called a module, ready for fast loading in the future.

The CMS command processor obtains input from three sources: that typed by the user at the machine console, that stored in an input stack, or that contained in a disk file used by the EXEC processor. This input, via the file system, selects a file to produce further input or to load and execute as a program module. When there is nothing left to do, CMS waits for more input from the user. Because all commands available to the user are stored as disk files and not built into the system, it is convenient, in an open-ended fashion, for a user to replace, change, or add commands.

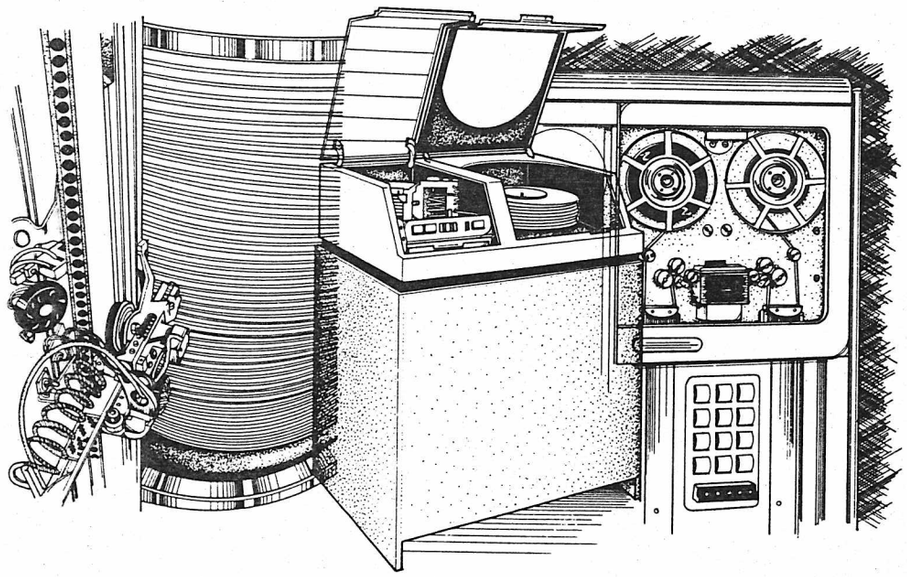
CMS was developed to support a particular type and style of work. It was designed to support its own development and maintenance. It is used to maintain the other components of VM/370 as well. Familiarity with previous work, as a designer and user, provided criteria to select or reject features based on ease of implementation, generality, and utility. In most cases, a subset of features was selected with the expectation of future work. We expected many operating systems to flourish in the virtual machine environment. What better place for experimentation with new system ideas? This has not been the case. Instead, many features were added to CMS to extend its usage into areas better served by new systems. A notable exception is RSCS, a system designed to control network communication from a dedicated machine.

Remote spooling and communications system

RSCS is an interrupt-driven, multi-tasking system that uses a dedicated computer with attached communications equipment for the support of data file transfer among computers and remote work stations. This system, developed after CP and CMS, has been described by its authors [18] in a way which illuminates sound concepts in virtual machine operating system design. RSCS is a paradigm which provides user service in the spirit of the CP/CMS concept. Operating in a virtual machine, it uses unit record equipment and communication links to send and receive files, in a store and forward fashion, to and from virtual machines, real machines, and remote work stations.

Within each VM/370 installation are many virtual machines identified by unique names defined within the CP directory. From a network viewpoint, CP appears as the central node of a star network with a virtual machine at each point. Information flows from a machine at the point, through CP at the center, and then back out to a point machine. So a file can be passed to someone by simply specifying the receiving machine's name and transmitting the file. This communication technique is only useful within one real system because the files are transmitted using each virtual machine's card reader and punch; it is impractical between real machines.

RSCS operates in a virtual machine which has attached telecommunications lines, channel to channel connections, and other computer communication devices. It



L. D. Stevens

The Evolution of Magnetic Storage

Since delivery of the first vacuum-column magnetic-tape transport in 1953 and the first movable-head disk drive in 1957, tape and disk devices in many configurations have been the principal means for storage of the large volumes of data required by data processing systems. Magnetic drums and other device geometries have also been important system components, but to a lesser extent. Over the past twenty-five years significant developments have been made that increase the capacity, reduce the cost, and improve the performance and reliability of these devices. With each improved device the range and nature of the applications undertaken have expanded and, in turn, led to a need for further device improvement. This paper gives a general review and historical perspective of magnetic storage development within IBM and is an introduction to the subsequent papers on disk, diskette, and tape technology and on disk manufacturing.

Introduction

Data processing applications of computers have grown over the past twenty-five years from incidental significance to a point where they have now become a pervasive influence in our society. Early data processing systems used magnetic tape as the principal storage medium for large data files. Processing was batch sequential on a job-by-job basis and the application focus was accounting. These systems had only secondary impact on the operational aspects of business. Those early computers are in sharp contrast to the data processing systems of today, which allow many different jobs to run concurrently with very-large-capacity on-line magnetic storage (*i.e.*, directly accessible without human intervention), data-base-oriented transaction processing, and an application focus on making more efficient use of operational resources.

Improvements in the cost, capacity, and performance of on-line magnetic storage have fueled these growing systems and their application capability. Three distinct periods can be identified in this evolution. During the first—the early years from 1953 to 1962—limited on-line storage was provided by the tape drives (with mounted reels of tape) attached to the system. Disk storage was a scarce resource, found only in those systems where the high cost, limited capacity, and difficulty of use could be justified by its capability for direct access of data. In the next period—the transition years from 1963 to 1966—rapid development of disk technology and systems software removed many of these constraints. Disk storage and on-line processing began to be an important part of most systems although tape storage and batch processing were still dominant. During the third period—the growth years from 1967 to 1980—the cost per Mbyte of disk storage was reduced twentyfold and with further improved systems software, new terminals, communication facilities, and on-line application development, substantial growth occurred in the on-line storage capacity of the average system; see Fig. 1. Here, the main memory capacity of the average IBM data processing system is compared with its disk and tape storage capacity during this period. Disk capacity per system increased by a factor of forty from a base of 23 Mbytes, attached-tape capacity increased by a factor of seven from a base of 47 Mbytes, and main-memory capacity per system increased by a factor of nineteen from a base of 50 Kbytes (where K = 1024). Disk capacity per system has been about 1000 times greater than that of main memory since 1973, and combined disk and tape capacity has grown to 1600 times that of main memory.

After a brief review of the basic capacity, cost and performance aspects of magnetic storage devices, this paper gives a historical perspective of device developments within IBM during each of these periods.

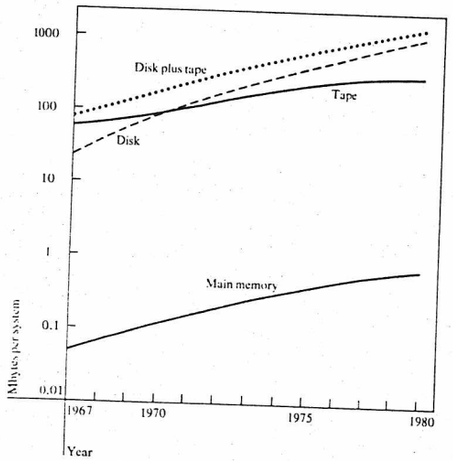


Figure 1 On-line storage capacity: disk, tape, and main memory capacity installed on the average IBM data processing system from 1967 to 1979.

Basic aspects of magnetic storage

Improvement of storage device cost, capacity, and performance has been achieved by a continuing development of magnetic recording and electromechanical access technology. Subsequent papers in this issue describe the innovative details of these developments for devices based on half-inch magnetic tape [1], rigid magnetic disks [2, 3], and flexible magnetic diskettes [4]. The following discussion provides a general introduction to these topics.

Magnetic recording technology includes the magnetic storage medium, the read and write heads, the recording channel electronics, the data encoding and clocking logic, and the technology that controls the head-to-medium spacing. The key parameter is areal density, the product of the linear bits per inch along a track (bpi) and the number of tracks per inch (tpi). It affects both capacity and performance and is the key parameter that determines the cost per Mbyte of storage.

Performance and capacity are also affected by access technology. For tape devices, it is the technology that controls the tape acceleration, velocity, and deceleration. The key parameters are the time required to start and stop, and the length of the resulting gap between blocks of data. For moving-head disk devices, it is the technology that controls the radial positioning of one or more read/write heads to selected concentric storage tracks. The key parameters are the average seek time (positioning time) and the accuracy of positioning.

- *Cost, cost/capacity, and cost/performance*

A large fraction of the cost of a tape transport or disk drive can be attributed to the fixed costs of the basic hardware required to support, protect, and transport the relatively inexpensive storage medium. The cost per Mbyte of storage is reduced with increasing areal-density capability because these fixed costs can be shared with more storage capacity. In addition, as areal-density capability has increased, the cost associated with the heads, actuators, media, recording channel electronics, and other technology components has remained nearly constant because of design improvements. Consequently, most designs have focused on providing substantial improvements in the cost per Mbyte for disks and the cost per Kbyte per second for tape by increasing the capacity and performance for equivalent or slightly increased cost.

- *Capacity*

Increased areal density has been the primary means of improving the capacity of both tape and disk devices. The companion papers [1-4] discuss the details of how this has been achieved through thinner particulate coatings having improved magnetic properties, the use of improved head materials, better fabrication techniques for smaller recording head gap length, reduced spacing between the head gap and the magnetic surface, and more accurate head positioning for disk drives.

Reducing the head-to-surface spacing has been a significant factor in increasing linear bit density. Contact between head and medium, with the proper head design, is acceptable for low-velocity diskette devices, as discussed by Engh [4]; contact is, however, not acceptable for high-velocity tape and disk devices. For these devices, a thin film of air is used to provide a lubricating air bearing that in turn determines the spacing. With magnetic tape and high-speed diskettes, the intrinsic boundary-layer airfilm forms a hydrodynamic air bearing as the flexible tape moves by the head. Control of this bearing is achieved by design of the head surface contour, as discussed by Engh [4] for diskettes, and by Harris, Phillips, Wells, and Winger [1] for half-inch magnetic tape. Spacing in the range of 5-10 microinches has been achieved. With magnetic disks, the spacing is accomplished by a separate air bearing support for the head. Progress in disk air bearing technology has reduced the spacing from 800 microinches in 1957, with hydrostatic (pressurized) air bearings, to the current 10-20 microinches today, with lightly loaded hydrodynamic (self-acting) slider bearings, as discussed by Harker, Brede, Pattison, Santana, and Taft [2]. These authors also discuss the evolution of improved disk read/write heads from laminated mu-metal, to ferrite, to thin films; the improvements in fabrication technologies that have reduced the gap length from 1000 to 40 microinches; and the improvements in particulate magnetic coating and processing technologies that have reduced magnetic film thickness on disks from 1200 to 20 microinches. The manufacturing aspects of these disk technologies are further discussed by Mulvany and Thompson [3].

Increased track density (tpi) has been of minor significance for half-inch tape devices. At first it was limited by head fabrication technology for the construction of seven, and later nine, accurately aligned parallel gaps across the half-inch tape. After the track pitch was established, the use of tape for system data interchange created a compatibility requirement that has constrained change. A similar data interchange constraint has influenced the design of diskettes, as discussed by Engh [4].

Track density improvement for moving-head disks has been an important contributor to increased areal density, but less significant than linear density. The track density of disks has been limited by the transverse resolution of the read/write head(s) and by the accuracy of their positioning. An error in positioning that exceeds a small fraction of the track width can cause either partial erasure of data on an adjacent track or failure to erase (or overwrite) previously written data. Either, upon readback, will result in a decreased signal-to-noise ratio. Mechanical detents were used to establish the final head position on early disks, and track density was limited to about 100 tpi. Separate write-wide/read-narrow or tunnel erase heads were required to establish guard bands between each data track. Positioning accuracy was improved by the development of closed-loop track-following servo systems. The result was a significant increase in tpi and the ability to use the same gap for reading and writing, as discussed by Harker *et al.* [2].

- *Performance*

The time required to obtain access to the start of a block of data plus the time required to transfer it to main memory determines the performance of both tape and disk devices. The relative importance of these two time components is a function of the data processing environment. The effective data rate—that fraction of the intrinsic data rate that can be realized—is of primary importance for either disk or tape used for sequential processing of a large number of records. But the rate of access (*i.e.*, accesses per second) deliverable with a reasonable response time is the important measure for disk storage used by interactive systems that generate essentially random requests for relatively short records.

Improvement of sequential processing performance has been significant. The intrinsic data rate of tape has been improved from 7.5 Kbytes per second (Kbyps) to 1.25 Mbytes per second (Mbyps) by means of increased linear density and velocity. The effective data rate of tape has been improved from about 50% of the intrinsic value to over 80% of the intrinsic value by increasing data block size and reducing the interblock-gap (IBG) length. Similar progress has been made in the sequential throughput from disks by improving the intrinsic data rate from 4.4 Kbyps to 3 Mbyps and in the effective data rate by developing of the multiple-head access structure, which provides a conceptual "cylinder" of data tracks at each access position and allows an essentially continuous stream of data by electronic switching between tracks.

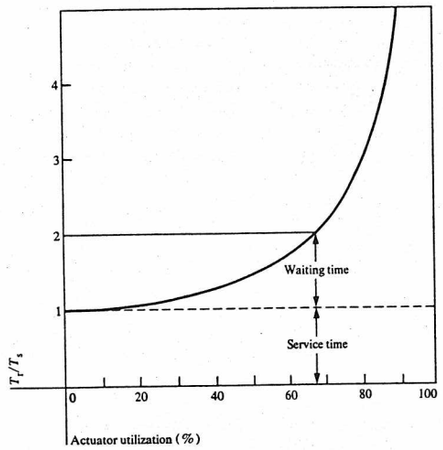


Figure 2 Disk response time and service time relationship: ratio of mean disk response time T_r to the average actuator service time T_s as a function of percent actuator utilization.

Comparison of the improvement in disk access rate achieved by actuator development leads to a discussion of the basic components of disk service time and the influence of actuator utilization on response time.

Disk service time is composed of three components: average seek time to position the heads, rotational latency to locate the start of the desired record (on the average, half of a revolution), and data transfer time (typically about one-tenth of a revolution). The maximum random access rate given by the reciprocal of this service time is not achievable when the constraint of a reasonable response time is imposed, because response time is composed not only of the service time but also of the time a request must wait while the actuator is servicing previous

requests. Waiting times become large when an actuator is working close to 100% of its capacity and grow without bound when saturation is reached. This is shown in Fig. 2, which plots the ratio of mean response time to the average service time as a function of actuator utilization, assuming random arrival of requests and constant service time [5, 6]. It can be seen that the waiting time component of the ratio begins to increase rapidly as the actuator is utilized more than two-thirds of the time. Here the ratio is two and the waiting time is equal to the service time. Given these assumptions, the random access throughput rate for disks has improved from about one access per second for the first disk drive to about 26 accesses per second for the most recently announced disk drive.

In addition to improvements in the access rate from individual disk actuators, the number of actuators that can be operating in parallel with overlapping seek, data transfer, and processing has been increased by improvements in the architecture of I/O subsystems [7], including the development of operating systems with multiprogramming capability, data management software, data channels, and intelligent device control units.

Historical perspective

This section reviews the development of magnetic storage devices and control units in the context of their systems software and application environment during each period of evolution.

- *The early years from 1953 to 1962*

Highlights of the changing environment during this period are as follows:

- Magnetic tape replaced punched cards as the principal storage medium for large data files; batch processing techniques continued to be used but at a much higher speed.
- Magnetic drums were used as main memory early in the period and as extensions of main memory throughout the period.
- Direct access to limited amounts of data became possible with the delivery of the first moving-head disk drives in 1957.
- Overlap of processing with I/O operations grew as the concepts of multiprogramming, CPU interrupt, data channels, and independent device control units were developed.

Magnetic tape (1953–1962) IBM delivered a high-performance digital magnetic tape drive in 1953 with the IBM 701. Its unique vacuum-column design isolated the high-inertia reel drive from a high-performance pinch roller and continuously rotating capstan used to control tape velocity [8]. The design concepts of this tape drive, the IBM 726, were the foundation for technological improvement for a decade.

Using an IBM-developed version of nonreturn-to-zero encoding (NonReturn-to-Zero-Inverted or NRZI), data recording was done on seven parallel tracks across a half-inch plastic substrate [acetate at first and later Mylar tape coated with magnetic iron oxide (Mylar is a trademark of E. I. du Pont de Nemours and Co., Inc., Wilmington, DE.)]. Each of six tracks represented a weighted bit in a six-bit binary coded decimal (BCD) character; the seventh was a redundancy bit. Recording density increased from 100 to 800 bpi during the period and tape velocity increased from 75 to 112.5 inches per second (ips), while the interblock gap (IBG) remained at 0.75 inch.

Improving start/stop time and thus reducing the necessary IBG length has been one objective of tape drive development. In addition, the blocking of a number of logical records to increase the size of each physical block between gaps has also been important to increased tape storage efficiency and effective data rate. In early systems, however, blocking was limited by the lack of memory for buffering and the lack of blocking and deblocking software. By the end of the period, main memory capacity had increased, and software had been developed to allow blocking factors of ten or more logical records per physical block. Blocking of logical records has continued to be an important factor relating to the efficiency and effective data rate of magnetic storage devices [9].

Processing of the data on magnetic tape followed a pattern similar to that of punched cards. Inputs were grouped together in a collection called a "batch," sorted into the same sequence as the files of data and processed job-by-job until the entire batch was completed.

To make space for new records and use the space of deleted records, the updating of a tape file required rewriting the entire tape. For example, transactions to a file would be read from one transport, the old file read from a second, and a new updated file written on a third. Therefore, the updatable unit for tape processing was a complete reel of tape.

This process was very efficient [10] if the file had a high activity rate. But if only a few percent of the records in a file were to be changed, as was typical of applications such as large life insurance policy files, maintenance of the master file was a major problem.

Another related problem was inquiry. The status of a record could be determined only through listings made each time the file was updated. Many applications, such as bank demand deposit and inventory control, require frequent inquiry, and the need for a storage medium with better access to individual records began to grow.

In early tape processing systems the CPU was not busy much of the time. It had to wait during access to each block of data and the writing of the new block. Much of the actual process time was spent inspecting records, with little useful work being accomplished. At first, the user had to program all of the I/O operations explicitly, including blocking and deblocking of records and recognition of special tape marks such as end-of-file and end-of-tape. Simple I/O monitor routines were soon developed, however, and stored on a system tape for common use [11]. In addition, tape control units were developed that independently managed the tape drive.

The IBM 777 Tape Record Coordinator, delivered in 1956 with the IBM 705 II, was the first such control unit and was also a precursor of the data channel. This buffered tape control unit allowed the CPU to overlap tape operation and processing by use of primitive "multiprogramming" [12]. Rochester, who was the first to use this term, describes the operation in an insurance policy file maintenance application [13]:

"The Tape Record Coordinator . . . runs its master tape units for the file maintenance job almost autonomously while passing over inactive records. Then when an active record is found, the calculator briefly interrupts the work it was doing (on another job) to deal with this active record. . . . When it is passing over inactive records the calculator needs to spend less than 1/2 percent of its time supervising the search. All the rest it can devote to the other job."

These primitive hardware and software concepts were expanded and generalized in later evolution periods and were destined to have a significant impact on the evolution of systems design as they were generalized to include all types of I/O including a person at a terminal [14].

Magnetic drums (1953–1962) Magnetic drums could not achieve as high a recording density as magnetic tape because of mechanical limitations of the head-to-medium spacing. A nominal spacing was established by machining with a final adjustment to about 0.001 inch by some type of differential screw. In addition to the spacing limitations, many early drums used a less efficient return-to-zero (RZ) recording technique to allow easy selective alteration of individual bits [15]. Linear densities were about 50 bpi and track spacing was about 20 tpi.

Two magnetic drums, typical of the early part of the period, were developed by IBM, one as an extension of electrostatic memory on the IBM 701 [16] and the other as main memory for the IBM 650 [17]. Use of magnetic drums as main memory began to diminish with the advent of magnetic cores, but their use as a main memory extension persisted well into the next period of evolution.

Toward the end of the period IBM developed an innovative drum for a version of the SAGE air defense computer. It was the first magnetic storage device to use a hydrodynamic slider bearing to establish head-to-medium spacing [18, 19]. The use of this technology permitted linear densities approaching those on tape with no mechanical adjustments.

Magnetic disks (1957–1962) Development of a hydrostatic air bearing to accurately space a magnetic head close to the surface of a rotating disk made possible the shipment in 1957 of the first movable-head disk drives with the IBM 305 [20] and 650 RAMAC systems. Its use allowed the magnetic head to follow minor axial runout of the disk while maintaining a constant spacing of about 800 microinches. A recording density of 100 bpi and 20 tpi [2 Kbits per square inch (Kbpsi)] was achieved [21]. This first disk drive used a pair of air-bearing-supported heads mounted on an access arm that could be moved under servo control to one of fifty 24-inch-diameter disks mounted on a vertical shaft and rotating at 1200 revolutions per minute (rpm). When positioned at a disk, the head pair could be moved radially to any of 100 tracks. Average seek time was 600 milliseconds (ms). Each track stored 500 BCD characters formatted into five fixed-length 100-character records. Each disk surface stored 50 000 characters; the 100 surfaces, a total of five million characters. Additional capacity was sacrificed for the simplicity of consistent track capacity and a single data rate; the maximum bpi was used only on the innermost track.

By the end of the period the hydrodynamic slider [22] had replaced the hydrostatic air bearing, and head-to-disk spacing in the range of 250 microinches was achievable. A comb of 50 access arms, one per disk surface, each with a slider and read/write head, was now made practical by the elimination of the large air compressor requirement for a similar array of hydrostatic bearings. Two such access structures, each with its own hydraulic actuator, were used, together with advanced magnetic technology, on the IBM 1301 disk drive, which was first installed in 1962. It provided 56 million characters of storage on fifty 24-inch-diameter disks. This capacity improvement of more than a factor of ten was achieved by an increased recording density of 520 bpi and 50 tpi (26 Kbpsi). The 1301 had

an improved average seek time of 165 ms, achieved by a high-performance hydraulic actuator and the elimination of the disk-to-disk motion.

Magnetic disks provided a combination of direct and sequential access. With the comb access mechanism a conceptual cylinder of 50 data tracks was formed at each track position. Once a position was selected by mechanical motion, any one of the 50 tracks could be selected by electronic switching. A data record on the selected track could be read on one disk revolution and an updated version written on the next revolution without affecting any other record. This in-place update and direct access capability of disk storage offered significant advantages.

Fixed-length records were used in these early disk drives. Record formats, however, were different for each application file and some flexibility was desirable. The first disk drive control unit, the IBM 7631, used a track associated with each cylinder of data for format control and allowed the specification of a different record format for each cylinder. It also provided for seek overlap in conjunction with the Input/Output Control System (IOCS). The IOCS provided for the management of buffering which allowed the use of multiple devices without the user being concerned with the synchronization of the hardware. The IOCS also blocked and deblocked the user's logical records to the physical tracks of the disk. It allowed a user to process a sequential data set from disk by use of high-level macro instructions.

The early disk storage devices presented other problems. These devices were not very reliable and there were no well-developed techniques to address a disk record directly, except in the very limited case that its physical address could be obtained by a linear transformation of the record identifier. As a result, the disk data management functions of the IOCS were, for the most part, an extension of those provided for tape. Direct-access processing required the application program to provide the physical address of the record (or block of records) by transformation of the logical record identifier to the physical device address before issuing calls to the IOCS. This required a detailed knowledge of the physical device. Soon, however, generalized indexing systems and nonlinear transformation techniques were developed to provide skip sequential and direct access processing [23, 24], and on-line data processing [20] began to grow [25].

With disk storage as an element of the system, jobs could be entered into the system in arrival sequence and queued on disk. If I/O units required by a job were busy, the processor could proceed with another task from the job queue on disk. The disk also provided residence for system software, allowing only the most frequently used programs to reside in main memory.

Disk storage remained a scarce system resource throughout the period because it was expensive. Although the cost per Mbyte had been reduced by a factor of two, disks were still limited to applications such as programming systems residence, reservations systems, and inventory control, where the cost of disk storage was still justified.

- *The transition years from 1963 to 1966*

Highlights of the changing environment during this period are as follows:

- The cost of a Mbyte of disk storage was reduced to approximately the cost of attached tape, and, as noted by Bonn [26], the design of computer systems entered a period of transition from tape to disk storage with on-line processing. Batch processing with tape, however, was still dominant.
- Introduction of the removable disk pack in 1963 provided off-line shelf storage of disks, but at the end of the period they were still twenty times more expensive than tape.
- IBM System/360 was introduced with an I/O architecture and with programming systems that required and enhanced disk storage capability.
- The role of magnetic tape began to shift from primary storage medium to systems interchange and archival storage.
- The first intelligent microprogrammed storage control units were introduced, and system attachment was simplified with the definition of a common I/O interface.

Magnetic tape (1963-1966) Two tape drives were announced with System/360. One, a new model of the IBM 7340 Hypertape drive [27], had a remarkable density (for the time) of 3022 bpi using one-inch tape in a cartridge with a two-cartridge autoloader, but was incompatible with existing half-inch tape libraries and was not widely accepted. The other, the IBM 2401, used a nine-track format for the eight-bit byte of System/360, was compatible with standard half-inch tape, and found wide acceptance. Recording density was at 800 bpi with NRZI encoding and later at 1600 bpi phase-encoded. Various models of the 2401 offered both nine- and seven-track formats. The interblock gap on nine-track models was 0.6 inch, but remained 0.75 inch on the seven-track models for compatibility.

Magnetic strip direct access storage Magnetic strip direct access devices were introduced in this period by IBM and others [28]. These devices offered very high capacity, moderate seek time, and low cost for both on-line and off-line storage. They suffered from high mechanical complexity compared to disk drives and were displaced by disk drives as disk storage capacity and cost improved.

Though differing in detail, the devices were all similar in that they consisted of a cartridge or cell containing a group of magnetic strips, each about twice the size of a punched card, one of which could be selected and rotated past a movable multitrack read/write head assembly. Spacing between the strip and head bar was determined by a film of air that created a hydrodynamic air bearing.

The IBM 2321 Data Cell Drive [29], delivered in 1966, had an on-line capacity of 400 Mbytes. It consisted of ten removable data cells, for off-line storage, and each data cell contained 200 13 × 2.25 × 0.005-inch Mylar strips with magnetic coating on one side and an antistatic carbon coating on the other. The cell was divided into twenty subcells and each of the ten strips in a subcell had a latching slot for picking the strip and a unique coding tab for selection. Each strip had chamfered edges and a "swallow" tail for control of anticlastic curvature and strip dynamics.

Recording density was 1750 bpi and 50 tpi on 100 data tracks on each strip. Strip seek time was 550 ms, and the selected strip was rotated at 1200 rpm to provide an intrinsic data rate of about 55 Kbyps and a maximum latency of 50 ms.

Magnetic disks (1963-1966) This was a very eventful period in the evolution of magnetic disk storage. The hydrodynamic slider developed for the 24-inch fixed-disk configuration led indirectly to the development of a small removable disk pack consisting of six 14-inch-diameter disks that stored 2.68 Mbytes. This innovation provided the first off-line storage capability for disks. The IBM 1311 disk drive [30] was first shipped in 1963 with the IBM 1441. Although a factor of two more expensive per Mbyte than attached tape and 60 times more expensive than tape off-line, it was a significant milestone, for it turned disk drive development in a fruitful new direction.

As the recording density improved from 1025 bpi and 50 tpi (51.25 Kbpsi) in 1963 to 2200 bpi and 100 tpi (220 Kbpsi) in 1966, the cost of disk storage was reduced to slightly less than attached tape, but still a factor of twenty more expensive than off-line tape. Average seek time was reduced by improvements in the hydraulic actuators from 150 to 75 ms. The disk rotational period was reduced from 57 to 25 ms and data rate increased from 69 to 312 Kbyps.

The first microprogrammed intelligent direct access storage control unit, the IBM 2841, was announced with System/360. It contained a processing unit with a transformer read-only storage for microcode control words. Through device-unique microcode, it could provide device-dependent interpretation of channel command words (CCWs) and provide real-time logical and electrical signaling to control devices with widely varying characteristics. It was attached to System/360 via a common (electrical and protocol) I/O interface and a data channel. This control unit design approach allowed implementation of the System/360 disk record format architecture [30] without separate logic circuits for each device type. In this architecture, called count-key-data (CKD), each record contains a count area and an optional key area as a header to the data record. The count area contains several fields of data; among them are the relative record number (on this track) and the length in bytes of the data record. The optional key field contains the record identifier and is used by the control unit to search for a record automatically, using its key as the argument.

Disk storage was no longer a scarce system resource after the delivery of the IBM 2314 File Facility in 1966. It had a new disk pack that stored 29 Mbytes. The packaging was innovative; it consisted of nine disk drives—eight on-line and one spare—and an improved version of the 2841 control unit housed in a single frame. The 233 Mbytes of on-line capacity and the relatively high performance that resulted from overlapping the seek time of its eight actuators [32] mark shipment of this file facility as the turning point for on-line system and application development [26].

New operating systems and data management software were developed in this period to use and enhance the growing capabilities of disks [33, 34]. These new complex systems required substantially more storage capacity than that provided by main memory and their development was made practical by direct access disk storage.

Data management capability [35, 36] provided by OS/360 introduced a level of device independence heretofore unavailable. A new set of access methods replaced IOCS. Two of these, Sequential Access Method (SAM) and Basic Direct Access Method (BDAM), were extensions of IOCS. The other two, Indexed Sequential Access Method (ISAM) and Basic Partitioned Access Method (BPAM), were innovative in that they handled for the user the translation of his logical record identifier into the physical address on disk.

System/360 data management also introduced space management for the storage subsystem. Disk storage

space in the past had been managed individually by each user or installation and it was possible for one user to destroy another's data by inadvertently writing over it. While the label processing provided by IOCS did preclude this to an extent, there was no central facility with which to determine the disposition of space within the storage subsystem.

Storage space was divided into subunits called "extents" by System/360. Data management allowed the user to request space in terms of records (or physical attributes such as tracks if he wished) which were then converted into physical extents by the data management routines. A volume table of contents (VTOC) which contained the disposition of all space on a volume (typically a disk pack)—used as well as free space—was recorded on each device.

The first widely accepted multiprogramming system was introduced with OS/360. With its capability for isolating programs from one another through use of memory protection, privileged instructions, priority interrupts, and application programs, as well as its ability to queue jobs for input and output, several unrelated jobs or tasks could occupy the same system. The objective of multiprogramming was to use all of the system resources as heavily as possible. There was little impact on a user if another unrelated program made use of disks when his task was in the compute state. Given several unrelated job streams, the relatively long access time to disk was masked and system throughput improved.

Disks now provided storage for the total on-line data base of systems programs, application programs, and application data files. As the transition from serial batch processing to on-line processing accelerated, file indexing and addressing techniques were improved, and the seeds of modern data base management systems were sown as techniques began to be developed [37] to use a common data base efficiently for a range of purposes.

- *The growth years from 1967 to 1980*

Highlights of the changing environment during this period include the following:

- The cost of a Mbyte of disk storage was reduced by more than a factor of twenty and on-line data processing became the dominant mode in most systems.
- Disk drive design returned to the early fixed-disk configuration: the use of removable disk packs began to diminish as the superior reliability of the new fixed-disk technology was proven and as transaction-oriented processing against the systems-managed data bases increased.
- Streaming tape drives were developed in response to a new role of tape in fixed-disk systems: tapes used as disk save/restore, in addition to systems interchange and archival storage.
- On-line mass storage using cartridges of wide tape in automatically managed libraries was introduced with on-line capacity greater than disks, lower costs than disks, but slower retrieval time.
- Small flexible disks or "diskettes" were introduced in 1971 for microprogram load and evolved into a new medium of system data interchange and the bulk storage medium for small computer systems.
- Smaller rigid disks about eight inches in diameter were introduced to provide low-cost on-line storage for small systems requiring moderate capacity, high performance, and small size.
- Intelligent storage control units were significantly improved in function and reduced in cost by the application of LSI microprocessors.
- Data base management software and new operating systems were introduced that were dependent on and allowed widespread use of disk storage in interactive data base applications.

Magnetic tape (1967–1980) Magnetic tape recording density increased from 1600 to 6250 bpi during this period. Maximum velocity increased from 112.5 to 200 ips and start/stop times improved with simpler mechanical designs. An innovative low-inertia high-torque motor driving a single capstan, with tape supplied from an additional "stubby" vacuum column, provided a new level of start/stop capability with the IBM 2420, first shipped in 1969. Tape velocity of 200 ips could be attained on this drive in less than 2 ms from a dead stop.

A model of the IBM 3420, shipped in 1973, provided further improvements in the drive that reduced the IBG to 0.3 inch, decreased the start/stop time to less than 1 ms, increased the linear bit density to 6250 bpi, and increased the intrinsic data rate to 1.25 Mbyps. This high bit density was achieved by the use of new recording technology and run-length-limited group-coded recording (GCR) encoding [1], while maintaining compatibility with existing tape [38].

Further simplification of the tape-drive mechanics was made possible by the elimination of the high-speed start/stop mechanism in a low-cost, low-performance, servo-controlled reel-to-reel tape drive announced in 1979 with the IBM 4331 and 8100 systems. This drive had two modes of operation: start/stop at 12.5 ips and streaming mode at 100 ips. In streaming mode the tape runs without stopping, provided the CPU can continue to accept the data. This mode of operation recognizes the importance of tape for the functions of disk save/restore as well as those of journaling, backup and recovery, and archival storage.

Mass storage systems (1975–1980) By the mid-1970s the management of very large magnetic tape libraries had become a major problem in terms of cost and space for installations with very large collections of data on tape. These library management costs in many cases exceeded the cost of tape and tape drives. This led to the need for automated tape library storage. Since the standard 10.5-inch tape reel was not convenient for automatic handling, a new medium—the IBM Data Cartridge—was shipped in 1975 with a Mass Storage System (MSS) [39, 40].

The data cartridge was about two inches in diameter and four inches long. It contained 770 inches of 2.7-inch-wide magnetic tape and could store 50 Mbytes. One of many cartridges stored in a honeycomb-like library could be automatically selected and transported in about 10–15 seconds to a data recording device, where the cartridge cover was removed and the tape wrapped around a mandrel containing a rotating read/write head. The tape is recorded in diagonal tracks called "stripes." A unique track-following servo was incorporated to locate a stripe and maintain its position relative to the read/write head.

The MSS combined the low cost of tape with the flexibility of disks by staging large blocks of data (250 Kbytes) on demand into disk storage where normal disk access was available. It thus provided virtual disk storage of from 32 to 472 Gbytes (gigabytes) of on-line data at a cost per Mbyte somewhat less than disks and with staging time in the range of 10–15 seconds.

Rigid magnetic disks (1967–1980) The low cost and the high-performance disk storage required by the IBM System/370 were provided by the development of an improved moving-head disk file (IBM 3330)—for data base storage—and a new fixed-head disk file (IBM 2305) replacing drum storage. Both of these files were announced in 1970 and each had a new microprogrammed control unit. These storage subsystems were also key components in the first widespread implementation of virtual storage techniques [41] announced on System/370 in 1972. In virtual storage systems each application considers itself the occupant of the addressable limits of the system, and pages or segments of data are automatically moved from disk storage to main memory by a combination of hardware and software so that only those segments actually in use will occupy main memory. These systems are heavily dependent on disk storage capacity and performance.

Disk pack capacity was increased from 29 to 100 Mbytes, using densities of 192 tpi and 4040 bpi (776 Kbpsi). Average seek time was reduced from 75 to 30 ms and disk rotational period from 25 to 16.7 ms. Improvements in flying height to 50 microinches and improved magnetic recording technology allowed the increase in bpi; development of the first track-following servo with a voice-coil motor provided the improved track density and seek time. This disk drive, the 3330-1, reduced the costs of on-line disk storage by nearly a factor of three and of off-line disk storage by a factor of two. They were both reduced another factor of two in 1974 when the 3330-11 was delivered with double the track density and double the capacity of the original 3330.

The heads in the fixed-head disk drive represented an advance in ferrite head and slider bearing technology. For the first time, the magnetic element and the slider were designed in an integrated structure using the same magnetic ferrite for the slider and the magnetic core [2]. Each slider contained nine read/write elements. Each of two different models of the IBM 2305 used multiple heads with a total of 768 of these elements: one model with each head on its own track, and one model with two heads per track. While they both had a rotation speed of 6000 rpm, the model with two heads per track reduced the average access time from 5 to 2.5 ms and reduced the storage capacity from 11.3 to 5.4 Mbytes. Data transfer rates were 1.5 and 2.0 Mbyps, respectively.

The two new microprogrammed control units had much in common and contained many design innovations [42] for improved performance, including a writable control memory. In addition to storing control microcode, this memory was used to buffer the count and key fields for error correction; to save counter contents, etc., for error logging and statistics; to accept multiple requests for data on the 2305 and execute them in the sequence of the shortest latency first; and, in conjunction with block multiplexer channels [43] and improved systems software, to allow the implementation of rotational position sensing (RPS). This innovation allowed the disk subsystem to monitor the progress of a data access and remain disconnected from the channel until just before the data passed under the read/write head, thus allowing a significant decrease in channel time required to locate a record.

Another innovation of the 3830 and the 2835 was a small flexible disk known as the 23FD [4] that was used in a read-only mode to load microcode into the writable control memory. Use of this little diskette was destined to grow beyond all expectations, as will be discussed.

The next step in disk evolution came with the development of a lightly loaded slider bearing [44] that carried the read/write head at a flying height of about 20 microinches,

eliminated the mechanism necessary to supply the 300-to-400-gram slider preload required by previous designs, and allowed starting and stopping while in contact with the disk. This slider was one of the innovations that led to the announcement of the IBM 3340 "Winchester" disk drive [45] in 1973. Removability of the disk pack in the conventional manner was not possible with these sliders because the sliders depended upon the disk to support them even when stopped: the disks and sliders had to stay together at all times. After some initial studies of techniques to load and unload the new low-mass heads from the disk, the decision was made to feature the capability of the head to start-stop in contact and its planned low cost. The complete head and disk assembly was made a removable unit. The IBM 3348 Data Module was developed incorporating heads, disks, spindle, head arms, and moving carriage. All of the key elements relating to critical tolerances associated with interchangeability of conventional disk packs were incorporated into the removable module—each head read only the data it had written. Significant savings were achieved by elimination of the head alignment procedures in manufacturing and the field [2]. The Data Module was sealed in a shock-mounted plastic enclosure that incorporated an access door, configured like a rolltop desk, for the drive actuator, air system, and electrical connection. Two heads (sliders) per surface reduced the stroke length required, and with improved servo technology [46], the average seek time was reduced from 30 to 25 ms.

The reliability of this new Winchester technology proved to be such an improvement that for the first time no scheduled maintenance was required for a disk drive. Other innovations in the 3340 worthy of note were the following: use of a single integrated circuit located on the access arm to provide the read/write electronics, automatic disk defect skipping, oriented magnetic particles for improved resolution of the magnetic medium, and a fixed-head feature as a part of the basic drive. Data module storage capacity of 35 Mbytes with two disks and 70 Mbytes with four disks was obtained by an areal density of 1.7 Mbpsi with 300 tpi and 5636 bpi.

The removability feature of disks diminished in importance as on-line capacity and reliability increased. Since the inception of the removable disk pack a trend toward fewer disk packs per drive had emerged. For example, with the 1311 the average number of disk packs per drive was greater than 12, with the 2314 it was down to 4, and with the 3330-11 only 1.2. These factors led to the development of a new disk drive with a nonremovable eight-disk spindle storing 317.5 Mbytes. First shipped in 1976, the IBM 3350 had a recording density of 3 Mbpsi (478 tpi and 6425 bpi) using improved Winchester technology. Average seek time was 25 ms. The cost per Mbyte of disk storage was reduced from that of the 3330-11 by a factor of two and by more than a factor of 70 from the original disk drive of twenty years previous.

Another factor of two reduction in disk cost per Mbyte was achieved with the announcement of the IBM 3370 in 1979. Film head technology, a new improved slider with flying height of less than 13 microinches, a new run-length encoding technique [2], and an improved track-following servo allowed 7.7 Mbpsi recording density. Average seek time of 20 ms was achieved with an innovative voice-coil actuator that used a single magnetic assembly for two independent actuators, each providing access to one-half of the data on a fixed seven-disk spindle containing 571 Mbytes of fixed-block storage [47, 48].

IBM announced another innovative disk drive in 1979 that used 210-mm disks (about eight inches). The IBM 3310 disk drive used a simple swing-arm actuator and had a unique track-following servo [49] that obtained its track-following position error information from the data head (by using samples taken between sectors of data) both for data recording and servo data detection.

Average seek time was reduced to 16 ms in 1980 with the announcement of the IBM 3380. Storage capacity was increased to 625 Mbytes per actuator and 1250 Mbytes per spindle by increased areal density. The rental cost of a Mbyte of disk storage was reduced to under one dollar per month.

A new LSI microprocessor [50–52] was at the heart of the new control unit, the IBM 3880, announced in 1979. This control unit provides both fixed-block and CKD (count-key-data) format control, significantly enhances maintenance facilities (through a special maintenance device) [53], and reduces the required number of channel reconnects by means of channel-command stacking. In addition, one model provides speed-match buffering from the high disk data rate to slower data channels.

Flexible magnetic disks (1967–1980) IBM shipped the first flexible disk drive in 1971 for use as a diagnostics and microprogram load device for the 3830 and 2835 Storage Control Units and the System/370 Model 145. The first drive, identified internally as 23FD, had a capacity of 81.6 Kbytes and was a read-only unit. Data and programs were written on a factory-controlled writer, and thus the tolerances involved in interchange between a large number of machines presented no problem. Interchange compatibility, however, was a major design consideration in all later flexible disk developments, as discussed by Engh [4].

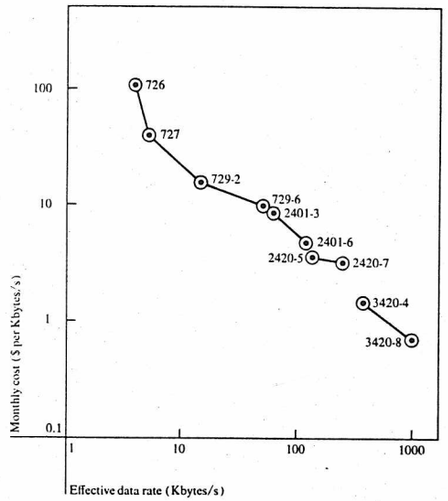


Figure 3 Magnetic tape cost and performance progress: Cost is the ratio of the monthly rental of one device to its intrinsic data rate, given in dollars per month per Kbyte/s. Performance is the effective data rate.

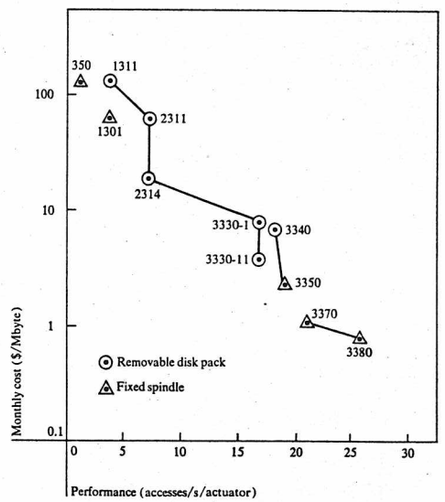


Figure 4 Magnetic disk cost and performance progress: Cost is the ratio of the monthly rental of one device to its capacity in Mbytes. Performance is the number of direct accesses (per actuator) per second deliverable with a device utilization of 67%.

The first diskette with write and read capability was the IBM 33FD. This drive, with a capacity of 243 Kbytes, was first shipped in 1973 as the output medium of the IBM 3740 data entry station. This increase in capacity was achieved through an increase in linear bit density from 1594 to 3268 bpi, an increase in the number of tracks from 32 to 77 (73 usable, 1 index, and 3 spares). Performance was also improved by increasing the speed of the disk from 90 to 360 rpm and the data rate from 33.3 to 250 Kbps. Track-to-track move time was reduced from 333 to 50 milliseconds per track, which reduced the average random access time (one-third of the tracks plus settle time) from 3.6 to 1.3 seconds.

The 33FD found wide acceptance in the industry, and since its introduction a large number of products which use the "floppy" disk have been announced by IBM and other suppliers. Zschu [54] makes the following observation on the importance of these devices:

"For large computer systems it means that keypunches, card handling equipment, key-to-tape and key-to-disk may be replaced with equipment using diskettes. For many minicomputer systems, it means that the cost of the peripherals, which today amount to the lion's share of the system cost, can be substantially reduced. Most importantly, it establishes . . . a medium that can be used as a multipurpose miniperipheral and which, when combined with LSI microprocessors, can do many general and special purpose data processing tasks at a fraction of their cost today."

With the announcement and shipment of the 43FD in 1976, the capability to record on both sides of the disk doubled the capacity to 568 Kbytes. In addition, this new drive reduced the track-to-track move time from 50 to 5 ms and the average random access time from 1.3 seconds to 170 ms. The capacity was again doubled in 1977 with the 53FD by means of a doubling of the linear bit density to 6418 bpi through use of improved heads and MFM encoding. The data rate was also doubled to 500 Kbps because of the increase in linear density. In 1979, the data rate was doubled again by increasing the disk speed to 720 rpm in the 72MD. This drive further increased the amount of on-line storage possible with diskettes by the addition of automatic diskette magazines. Each magazine contained ten diskettes and each drive, two magazines, for an on-line storage capacity of about 24 Mbytes.

Since their introduction in 1973, the read/write capacity of IBM diskettes has been increased by a factor of fifteen, the data rate by a factor of thirty, and the average seek

time reduced by nearly a factor of eight. Engh [4] describes the innovations in air bearing control, magnetic head design, electronic circuits, and mechanical design that have permitted these substantial improvements.

Summary

IBM has developed several magnetic storage products that were the first of their kind in that they provided a significant new functional capability for data processing systems. Progress in the evolution of each of these products can be identified with one or more technological innovations which go beyond improvements strictly in basic magnetic recording technology. These include

- The vacuum column tape drive
  - High-torque, low-inertia motor
  - Stubby vacuum column
  - Encoding techniques: NRZI and run-length-limited codes
- The moving-head disk file
  - Hydrostatic air bearing and hydrodynamic slider bearing
  - Mechanical design for disk pack removability
  - Voice-coil motor and track-following servos
  - Data encoding, detection, and clocking circuits
- The flexible diskette
  - Low-cost actuator design
  - Low-wear in-contact head design
  - Diskette materials and packaging for ease of use
- The magnetic cartridge mass store
  - Rotating-head design for digital recording
  - Flexible-media track-following servos
  - Cartridge library storage facility
  - Data staging algorithms

Figures 3 and 4 summarize graphically the overall progress in tape and disk cost and performance. They are plots in cost-performance space of the key devices discussed in the body of the paper and in the subsequent papers on tape and disk storage technology. The points connected by lines are members of the same product family and show the improvements made on a similar technological base.

In addition to the specific developments cited in this paper, some general references are included [55-67] for background on the thread of development of the technologies.

Acknowledgments

The author gratefully acknowledges the assistance of several individuals who contributed to the preparation of this paper. Helpful comments on the contents and organization were made by C. J. Bashe, J. T. Engh, J. M. Harker, J. P. Harris, F. Kurzweil, R. B. Mulvany, M. I. Schor, and R. W. Shomler. C. R. Holleran contributed to the discussion of the disk control units and A. J. Bonner to the sections concerned with systems and data management software development.

References

1. J. P. Harris, W. B. Phillips, J. F. Wells, and W. D. Winger, "Innovations in the Design of Magnetic Tape Subsystems," *IBM J. Res. Develop.* **25**, 691-699 (1981, this issue).
2. J. M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana, and L. G. Taft, "A Quarter Century of Disk File Innovation," *IBM J. Res. Develop.* **25**, 677-689 (1981, this issue).
3. R. B. Mulvany and L. H. Thompson, "Innovations in Disk File Manufacturing," *IBM J. Res. Develop.* **25**, 711-723 (1981, this issue).
4. James T. Engh, "The IBM Diskette and Diskette Drive," *IBM J. Res. Devleop.* **25**, 701-710 (1981, this issue).
5. H. Kobayashi, *Modeling and Analysis—An Introduction to System Performance Evaluation Methodology*, Addison-Wesley Publishing Co., Reading, MA, 1978.
6. P. H. Seaman, R. A. Lind, and T. L. Wilson, "On Teleprocessing System Design, Part IV: An Analysis of Auxiliary Storage Activity," *IBM Syst. J.* **5**, 158-170 (1966).
7. J. P. Buzen, "I/O Subsystem Architecture," *Proc. IEEE* **63**, 871-879 (1975).
8. W. S. Buslik, "IBM Magnetic Tape Reader and Recorder," *Proceedings of the Joint AIEE-IRE-ACM Computer Conference (Review of Input and Output Equipment Used in Computer Systems)*, March 1953, pp. 86-90.
9. C. W. Bachman, "The Evolution of Storage Structures," *Commun. ACM* **15**, 628-634 (1972).
10. V. P. Turnburke, Jr., "Sequential Data Processing Design," *IBM Syst. J.* **2**, 37-48 (1963).
11. R. F. Rosin, "Supervisory and Monitor Systems," *ACM Computing Surv.* **1**, 37-54 (1969).
12. E. F. Codd, "Multiprogramming," *Advances in Computers*, Vol. 3, Academic Press, Inc., New York, 1962, pp. 77-153.
13. N. Rochester, "The Computer and its Peripheral Equipment," *Proceedings of the Eastern Joint Computer Conference*, Boston, MA, 1955, pp. 64-69.
14. M. V. Wilkes, "Historical Perspective—Computer Architecture," *AFIPS Conf. Proc., Fall Joint Computer Conf.* **41**, 971-976 (1972).
15. A. A. Cohen and W. R. Keye, "Selective Alteration of Digital Data in a Magnetic Drum Computer," Contract N6ONR-240 Task 1: Engineering Research Assoc., Office of Naval Research: December 1947.
16. C. E. Frizzell, "Engineering Description of the IBM Type 701 Computer," *Proc. IRE* **41**, 1257-1287 (1953).
17. F. E. Hamilton and E. C. Kubie, "The IBM Magnetic Drum Calculator Type 650," *J. ACM* **1**, 13-20 (1954).
18. T. Leverett, "Floating Drum Head With Retractable Shoe," *Technical Report PH23-000121M-306*, IBM Corporation, Kingston, NY, 1957.
19. W. A. Stetler, V. G. Spiegel, and R. C. Braen, "Vertical Magnetic Drum With Floating Head," *Technical Report PH23-08001TR-330*, IBM Corporation, Kingston, NY, 1959.
20. M. L. Lesser and J. W. Haanstra, "The Random-Access Memory Accounting Machine—I. System Organization of the IBM 305," *IBM J. Res. Develop.* **1**, 62-71 (1957).
21. T. Noyes and W. E. Dickinson, "The Random-Access Memory Accounting Machine—II. The Magnetic-Disk, Random-Access Memory," *IBM J. Res. Develop.* **1**, 72-75 (1957).
22. R. K. Brunner, J. M. Harker, K. E. Haughton, and A. G. Osterlund, "A Gas Film Lubrication Study, Part III, Experimental Investigation of Pivoted Slider Bearings," *IBM J. Res. Develop.* **3**, 260-274 (1959).
23. W. W. Peterson, "Addressing for Random Access Storage," *IBM J. Res. Develop.* **1**, 130-146 (1957).
24. Werner Buchholz, "File Organization and Addressing," *IBM Syst. J.* **2**, 86-111 (1963).
25. J. D. Aron, "Information Systems in Perspective," *ACM Computing Surveys* **1**, 213-236 (1969).
26. T. H. Bonn, "Mass Storage: A Broad Review," *Proc. IEEE* **54**, 1861-1870 (1966).
27. R. A. Barbeau and J. I. Aweida, "IBM 7340 Hypertape Drive," *AFIPS Conf. Proc., Fall Joint Computer Conf.* **24**, 591-602 (1963).
28. K. R. London, *Techniques for Direct Access*, Auerbach, Philadelphia, 1973, pp. 70-77.
29. A. F. Shugart and Y. H. Tong, "IBM 2321 Data Cell Drive," *AFIPS Conf. Proc., Spring Joint Computer Conf.* **28**, 335-345 (1966).
30. J. D. Carothers, R. K. Brunner, J. L. Dawson, M. O. Halfhill, and R. E. Kubec, "A New High Density Recording System: The IBM 1311 Disk Storage Drive With Interchangeable Disk Packs," *AFIPS Conf. Proc., Fall Joint Computer Conf.* **24**, 327-340 (1963).
31. "Introduction to IBM Direct Access Storage Devices and Organization Methods," *Publication No. GC20-1649*, IBM Corporation, San Jose, CA.
32. J. Abate, H. Dubner, and S. B. Weinberg, "Queueing Analysis of the IBM 2314 Disk Storage Facility," *J. ACM* **15**, 557-589 (1968).
33. C. W. Bachman and S. B. Williams, "A General Purpose Programming System for Random Access Memories," *AFIPS Conf. Proc., Fall Joint Computer Conf.* **26**, 411-422 (1964).
34. C. B. Poland, "Advanced Concepts of Utilization of Mass Storage," *Proc. IFIPS Congress 65* **1**, 249-254 (1965).
35. W. A. Clark, "The Functional Structure of OS/360, Part III: Data Management," *IBM Syst. J.* **5**, 30-51 (1966).
36. G. G. Dodd, "Elements of Data Management," *ACM Computing Surveys* **1**, 117-132 (1969).
37. J. Martin, *Computer Data Base Organization*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.
38. J. A. Rodriguez, "An Analysis of Tape Drive Technology," *Proc. IEEE* **63**, 1153-1159 (1975).
39. C. T. Johnson, "The IBM 3850: A Mass Storage System with Disk Characteristics," *Proc. IEEE* **63**, 1166-1170 (1975).
40. J. P. Harris, R. S. Rohde, and N. K. Arter, "IBM 3850 Mass Storage System: Design Aspects," *Proc. IEEE* **63**, 1171-1176 (1975).
41. P. J. Denning, "Virtual Memory," *ACM Computing Surveys* **2**, 153-189 (1970).
42. G. R. Ahearn, Y. Dishon, and R. N. Snively, "Design Innovations of the IBM 3830 and 2835 Storage Control Unit," *IBM J. Res. Develop.* **16**, 11-18 (1972).
43. D. T. Brown, R. L. Eibsen, and C. A. Thorn, "Channel and Direct Access Device Architecture," *IBM Syst. J.* **11**, 186-199 (1972).
44. M. W. Warner, "Flying Magnetic Transducer Assembly Having Three Rails," U.S. Patent No. 3,823,416, July 9, 1974.
45. R. B. Mulvany, "Engineering Design of a Disk Storage Facility with Data Modules," *IBM J. Res. Develop.* **18**, 489-505 (1974).
46. R. K. Oswald, "Design of a Disk File Head-Positioning Servo," *IBM J. Res. Develop.* **18**, 506-512 (1974).
47. A. D. Rizzi and J. S. Makiyama, "The IBM 3370 Direct Access Storage," *Disk Storage Technology*, 31-33 (1980); Order No. GA26-1665-0, available through IBM branch offices.
48. D. L. Nelson, "The Format of Fixed Block Architecture in the IBM 3370 DAS," *op. cit.* Ref. 47, pp. 34-35.
49. R. D. Commander and J. R. Taylor, "Servo Design for an Eight-Inch Disk File," *op. cit.* Ref. 47, pp. 89-97.
50. J. I. Norris, "A High Performance Microprocessor," *op. cit.* Ref. 47, pp. 27-30.
51. H. Bardsley, "The IBM 3880 Microprocessor Control," *op. cit.* Ref. 47, pp. 69-72.
52. C. R. Holleran, "An Overview of the IBM 3880 Storage Control," *op. cit.* Ref. 47, pp. 62-64.
53. S. J. Duchak, "Maintenance of the IBM 3880 Storage Control," *op. cit.* Ref. 47, pp. 78-82.
54. E. V. W. Zschau, "The IBM Diskette and Its Implications for Minicomputer Systems," *Computer* **6**, No. 6, 21-26 (1973).
55. F. P. Brooks, "Mass Memory in Computer Systems," *IEEE Trans. Magnetics* **MAG-5**, 635-639 (1969).
56. E. W. Pugh, "Storage Hierarchies; Gaps, Cliffs, and Trends," *IEEE Trans. Magnetics* **MAG-7**, 810-814 (1971).
57. R. E. Matick, "Review of Current Proposed Technologies for Mass Storage Systems," *Proc. IEEE* **60**, 266-289 (1972).
58. J. M. Harker and Hsu Chang, "Magnetic Disks for Bulk Storage—Past and Future," *AFIPS Conf. Proc., Spring Joint Computer Conf.* **40**, 945-955 (1972).
59. F. G. Withington, "Five Generations of Computers," *Harvard Bus. Rev.* **52**, 99-103 (1974).
60. U. O. Gagliardi, "Trends in Computer System Architecture," *Proc. IEEE* **63**, 858-862 (1975).
61. K. E. Haughton, "An Overview of Disk Storage Systems," *Proc. IEEE* **63**, 1148-1152 (1975).
62. C. D. Mee, "A Comparison of Bubble and Disk Storage Technologies," *IEEE Trans. Magnetics* **MAG-12**, 1-6 (1976).
63. A. S. Hoagland, "Magnetic Recording Storage," *IEEE Trans. Computers* **C-25**, 1283-1288 (1976).
64. R. E. Matick, *Computer Storage Systems and Technology*, John Wiley & Sons, Inc., New York, 1977.
65. N. C. Wilhelm, "A General Model for the Performance of Disk Systems," *J. ACM* **24**, 14-31 (1977).
66. A. S. Hoagland, "Storage Technology: Capabilities and Limitations," *Computer* **12**, No. 5, 12-18 (1979).
67. A. S. Hoagland, "Trends and Projections in Magnetic Recording Storage on Particulate Media," *IEEE Trans. Magnetics* **MAG-16**, 26-29 (1980).

Received April 3, 1980; revised November 3, 1980

The author is located at the IBM General Products Division laboratory, San Jose, California 95150.